HP NonStop SQL/MX Release 3.2.1 Reference Manual

Abstract

This manual describes the syntax of SQL language elements—data types, expressions, functions, identifiers, literals, and predicates—and SQL statements of HP NonStop™ SQL/MX, the NonStop relational database management system based on ANSI SQL:1999. The manual also includes embedded SQL statements and MXCI commands.

Product Version

NonStop SQL/MX Release 3.2.1

Supported Release Version Updates (RVUs)

This publication supports J06.15 and all subsequent J-series RVUs and H06.26 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
691117-004	November 2013

Document History

Part Number	Product Version	Published
640322-001	NonStop SQL/MX Release 3.0	February 2011
663850-001	NonStop SQL/MX Release 3.1	October 2011
691117-001	NonStop SQL/MX Release 3.2	August 2012
691117-002	NonStop SQL/MX Release 3.2.1	February 2013
691117-003	NonStop SQL/MX Release 3.2.1	July 2013
691117-004	NonStop SQL/MX Release 3.2.1	November 2013

Legal Notices

Printed in the US

HP NonStop SQL/MX Release 3.2.1 Reference Manual

Index | Figures | Tables

1. Introduction

2. SQL/MX Statements

3. Embedded-Only SQL/MX Statements

6. SQL/MX Language Elements

7. SQL/MX Clauses

9. SQL/MX Functions and Expressions

E. Standard SQL and SQL/MX

Figures

Tables

What’s New in This Manual

Manual Information

Abstract

This manual describes the syntax of SQL language elements—data types, expressions, functions, identifiers, literals, and predicates—and SQL statements of HP NonStop™ SQL/MX, the NonStop relational database management system based on ANSI SQL:1999. The manual also includes embedded SQL statements and MXCI commands.

Product Version

NonStop SQL/MX Release 3.2.1

Supported Release Version Updates (RVUs)

This publication supports J06.14 and all subsequent J-series RVUs and H06.25 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
691117-004	November 2013

Document History

Part Number	Product Version	Published
640322-001	NonStop SQL/MX Release 3.0	February 2011
663850-001	NonStop SQL/MX Release 3.1	October 2011
691117-001	NonStop SQL/MX Release 3.2	August 2012
691117-002	NonStop SQL/MX Release 3.2.1	February 2013
691117-003	NonStop SQL/MX Release 3.2.1	July 2013
691117-004	NonStop SQL/MX Release 3.2.1	November 2013

New and Changed Information

Changes to 691117-004 manual:

Changed the existing content for MXCI Command on page 4-55.

Changes to 691117-003 manual:

Moved the following utilities from the chapter, SQL/MX Utilities on page 5-1 to the chapter, SQL/MX Statements on page 2-1:

- DOWNGRADE Utility on page 2-175
- DUP Utility on page 2-194
- FASTCOPY Utility on page 2-226
- MODIFY Utility on page 2-271
- POPULATE INDEX Utility on page 2-304
- PURGEDATA Utility on page 2-307
- RECOVER Utility on page 2-311
- UPGRADE Utility on page 2-412

Updated the MODIFY utility syntax in Manage Partitions of Range Partitioned Tables and Indexes on page 2-274 and Manage Partitions of Hash Partitioned Tables and Indexes on page 2-281

Updated Considerations for GIVE SCHEMA on page 2-239

Changes to 691117-002 manual:

Added a note for 64-bit application support in SQL Descriptor Area Data Type Declarations of GET DESCRIPTOR on page 3-51 and

SQL Item Descriptor Area of SET DESCRIPTOR on page 3-79

Added the following for sequence generator support:

ALTER SEQUENCE Statement on page 2-13

CREATE SEQUENCE Statement on page 2-100

DROP SEQUENCE Statement on page 2-185

Gaps in sequence generator values on page 2-135

Managing a Sequence Generator on page 2-288

Offline Partition Management for Sequence Generators on page 2-292

Sequence Generators on page 6-110

Pseudocolumns on page 6-105

Updated the following for sequence generator support:

GRANT Statement on page 2-240

REVOKE Statement on page 2-317

GIVE Object Statement on page 2-237

Considerations for SHOWDDL on page 4-85

CLEANUP Operation on page 5-3

FIXUP Operation on page 5-8

Renaming Guardian Location of Partitions of Tables, Indexes or Sequence Generators on page 2-289

SEQUENCE_GENERATORS Table on page 10-28

Changes to 691117-001 manual:

- Statements:
  - Updated REVOKE Statement on page 2-317.
  - Updated ALTER VIEW Statement on page 2-49.
  - Updated CREATE TABLE Statement on page 2-107.
  - Updated CREATE VIEW Statement on page 2-154 and CREATE TRIGGER Statement on page 2-144.
  - Updated DELETE Statement on page 2-162.
  - Updated DROP SCHEMA Statement on page 2-183.
  - Updated INSERT Statement on page 2-252.
  - Updated UPDATE Statement on page 2-385.
- Embedded-only SQL/MX statements:
  - GET DIAGNOSTICS Statement on page 3-55.
- MXCI Commands
  - Updated SHOWDDL Command on page 4-83.
  - Added SHOWSTATS Command on page 4-112.
- Utilities:
  - Added CLEANUP Operation on page 5-3.
  - Updated DOWNGRADE Utility on page 2-175.

- Updated MODIFY Utility on page 2-271.
- Updated FIXUP Operation on page 5-8.
- Updated FIXRCB Operation on page 5-7.
- Updated PURGEDATA Utility on page 2-307.
- Updated UPGRADE Utility on page 2-412.
- Updated VERIFY Operation on page 5-79.

- Functions and Expressions:
  - DATE_ADD Function on page 8-44.
  - DATEADD Function on page 8-44.
  - DATE_SUB Function on page 8-48.
  - DATEDIFF Function on page 8-46.
- Metadata Tables:
  - Updated the SCHEMATA Table on page 10-10
  - Updated the OBJECTS Table on page 10-22.
  - Updated SEQUENCE_GENERATORS Table on page 10-28.
  - Updated VWS Table on page 10-35.
  - Added the CQD DDL_VIEW_SIMILARITY_CHECK to Query Optimization and Performance on page 10-66

About This Manual

This manual describes the syntax of SQL language elements—data types, expressions, functions, identifiers, literals, and predicates—and SQL statements of NonStop SQL/MX, the NonStop relational database management system based on ANSI SQL:1999. The manual also includes embedded SQL statements and MXCI commands.

Note. In this manual, SQL language elements, statements, and clauses within statements that are extensions to the ANSI SQL:1999 standard are noted as SQL/MX extensions.

Audience

This manual is intended for database administrators and application programmers who are using NonStop SQL/MX through MXCI—the SQL/MX conversational interface—or as embedded SQL, ODBC, or JDBC applications to access databases. To use this product, the reader must be familiar with Structured Query Language (SQL) and with American National Standard Database Language SQL:1999.

Organization

Section	Description
Section 1, Introduction	Introduces NonStop SQL/MX and covers topics such as entering statements and commands in MXCI, database security, data consistency and integrity, transaction management, querying SQL/MP databases, and ANSI compliance and SQL/MX extensions.
Section 2, SQL/MX Statements	Describes the SQL statements supported by NonStop SQL/MX.
Section 3, Embedded-Only SQL/MX Statements	Describes the SQL statements that you embed only in programs.
Section 4, MXCI Commands	Describes the MXCI commands that you run only in the SQL/MX conversational interface (MXCI).
Section 5, SQL/MX Utilities	Describes utilities that perform such tasks as duplicating files, importing data, migrating metadata, and populating indexes.
Section 6, SQL/MX Language Elements	Describes parts of the language, such as database objects, data types, expressions, identifiers, literals, and predicates, which occur within the syntax of SQL/MX statements and MXCI commands.
Section 7, SQL/MX Clauses	Describes clauses used by SQL/MX statements.

Section	Description
Section 9, SQL/MX File Attributes	Describes SQL/MX file attributes.
Section 8, SQL/MX Functions and Expressions	Describes specific functions and expressions that you can use in SQL/MX statements.
Section 10, Metadata Tables	Describes the user metadata tables in NonStop SQL/MX, including system defaults.
Appendix A, Quick Reference	Is a quick reference to commands, statements, and utilities.
Appendix B, Reserved Words	Lists the words that are reserved in NonStop SQL/MX.
Appendix C, Limits	Describes limits in NonStop SQL/MX.
Appendix D, Sample Database	Describes the schema and tables of the sample database, which is the basis for many examples in this manual and other SQL/MX manuals.
Appendix E, Standard SQL and SQL/MX	Describes how NonStop SQL/MX conforms to the ANSI standard.

Related Documentation

This manual is part of the HP NonStop SQL/MX library of manuals, which includes:

Introductory Guides

SQL/MX Comparison Guide for SQL/MP Users	Describes SQL differences between NonStop SQL/MP and NonStop SQL/MX.
SQL/MX Quick Start	Describes basic techniques for using SQL in the SQL/MX conversational interface (MXCI). Includes information about installing the sample database.

Reference Manuals

SQL/MX Reference Manual	Describes the syntax of SQL/MX statements, MXCI commands, functions, and other SQL/MX language elements.
SQL/MX Messages Manual	Describes SQL/MX messages.
SQL/MX Glossary	Defines SQL/MX terminology.

Installation Guides

SQL/MX Installation and Upgrade Guide	Describes how to plan for, install, create, and upgrade an SQL/MX database. Explains how to install and upgrade to a higher version of SQL/MX.
NSM/web Installation Guide	Describes how to install NSM/web and troubleshoot NSM/web installations.

Connectivity Manuals

SQL/MX Connectivity Service Manual	Describes how to install and manage the HP NonStop SQL/MX Connectivity Service (MXCS), which enables applications developed for the Microsoft Open Database Connectivity (ODBC) application programming interface (API) and other connectivity APIs to use NonStop SQL/MX.
SQL/MX Connectivity Service Administrative Command Reference	Describes the SQL/MX administrative command library (MACL) available with the SQL/MX conversational interface (MXCI).
ODBC/MX Driver for Windows	Describes how to install and configure HP NonStop ODBC/MX for Microsoft Windows, which enables applications developed for the ODBC API to use NonStop SQL/MX.

Migration Guides

SQL/MX Installation and Upgrade Guide	Describes how to plan for, install, create, and upgrade an SQL/MX database. Explains how to install and upgrade to a higher version of SQL/MX.
HP NonStop SQL/MP to SQL/MX Database and Application Migration Guide	Describes how to migrate databases and applications from SQL/MP to SQL/MX.
NonStop NS-Series Database Migration Guide	Describes how to migrate NonStop SQL/MX, NonStop SQL/MP, and Enscribe databases and applications to HP Integrity NonStop NS-series systems.

Data Management Guides

SQL/MX Management Manual	Describes how to manage an SQL/MX database.
SQL/MX Data Mining Guide	Describes the SQL/MX data structures and operations to carry out the knowledge-discovery process.
SQL/MX Report Writer Guide	Describes how to produce formatted reports using data from an SQL/MX database.
DataLoader/MX Reference Manual	Describes the features and functions of the DataLoader/MX product, a tool to load SQL/MX databases.

Application Development Guides

SQL/MX Programming Manual for C and COBOL	Describes how to embed SQL/MX statements in ANSI C and COBOL programs.
SQL/MX Query Guide	Describes how to understand query execution plans and write optimal queries for an SQL/MX database.
SQL/MX Queuing and Publish/Subscribe Services	Describes how NonStop SQL/MX integrates transactional queuing and publish or subscribe services into its database infrastructure.
SQL/MX Guide to Stored Procedures in Java	Describes how to use stored procedures that are written in Java within NonStop SQL/MX.

Online Help

Reference Help	Overview and reference entries from the *SQL/MX Reference Manual*.
Messages Help	Individual messages grouped by source from the *SQL/MX Messages Manual.*
Glossary Help	Terms and definitions from the *SQL/MX Glossary*.
NSM/web Help	Context-sensitive help topics that describe how to use the NSM/web management tool.
Visual Query Planner Help	Context-sensitive help topics that describe how to use the Visual Query Planner graphical user interface.
SQL/MX Database Manager Help	Contents and reference entries from the *SQL/MX Database Manager User Guide*.

The NSM/web, SQL/MX Database Manager, and Visual Query Planner help systems are accessible from their respective applications. You can download the Reference, Messages, and Glossary online help from the HP Software Depot at http://www.software.hp.com. For more information about downloading the online help, see the *SQL/MX Release 3.2 Installation and Upgrade Guide*.

These manuals are part of the SQL/MP library of manuals and are essential references. For more information about SQL/MP Data Definition Language (DDL) and SQL/MP installation and management, see the following SQL/MP manuals:

Related SQL/MP Manuals

SQL/MP Reference Manual	Describes the SQL/MP language elements, expressions, predicates, functions, and statements.
SQL/MP Installation and Management Guide	Describes how to plan, install, create, and manage an SQL/MP database. Describes installation and management commands and SQL/MP catalogs and files.

Notation Conventions

Icons

These icons appear in the left margins of this manual. Each icon represents a specific context of the SQL/MX syntax and semantics:

Embed	Designates information that is specific to embedding SQL/MX statements in programs. This information applies to any of the supported programming languages.
C/COBOL	Designates information that is specific to embedding SQL/MX statements in C or COBOL programs.
Java	Designates information that is specific to embedding SQL/MX statements in Java programs.
MXCI	Designates information that is specific to the SQL/MX conversational interface (MXCI).
Pub/Sub	Designates information that is specific to queuing and publish or subscribe services.

The ■ symbol represents the end of context-specific information in one or more paragraphs or in a syntax diagram.

Hypertext Links

A blue underline is used to indicate a hypertext link within text. Clicking a passage of text with a blue underline takes you to the location described. For example, this requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

The following list summarizes the notation conventions used for syntax presentation in this manual:

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

`file-name`

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

`pathname`

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name
```

```
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]
K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }
```

```
ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]…
```

```
[ - ] {0|1|2|3|4|5|6|7|8|9}…
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char…"
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;
```

```
LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must enter as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, there are no spaces permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]...
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to docsfeedback@hp.com.

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

1 Introduction

NonStop SQL/MX allows you to use SQL/MX DML statements, which comply closely to ANSI SQL:1999, to access SQL/MP and SQL/MX databases.

This introduction describes:

Other sections of this manual describe the syntax and semantics of individual statements, commands, and language elements.

SQL/MX Language

The SQL/MX language consists of statements, commands, and other language elements that you can use to access SQL/MP and SQL/MX databases. For more information on the SQL/MP language, see the *SQL/MP Reference Manual*.

You can run SQL/MX statements from the SQL/MX conversational interface, MXCI, or embed SQL/MX statements in programs written in C, C++, COBOL, or Java. For more information on MXCI, see MXCI SQL/MX Conversational Interface on page 1-2. For descriptions of individual SQL/MX statements, see Section 2, SQL/MX Statements.

Some SQL/MX statements can be used only within embedded SQL programs and cannot be run in an MXCI session. For descriptions of these statements, see Section 3, Embedded-Only SQL/MX Statements. For more information on embedding SQL/MX statements in programs, see the *SQL/MX Programming Guide for C and COBOL*.

MXCI commands are SQL/MX extensions that typically affect attributes of an MXCI session. These commands can be run only in MXCI, with a few exceptions. For more information, see Section 4, MXCI Commands.

SQL/MX language elements are part of statements and commands and include data types, expressions, functions, identifiers, literals, and predicates. For more information, see Section 6, SQL/MX Language Elements. For more information on specific functions and expressions, see Section 8, SQL/MX Functions and Expressions.

User metadata tables, such as the SYSTEM_DEFAULTS table, histogram tables, and other tables, contain SQL/MX metadata that the user rather than the system maintains. For more information, see Section 10, Metadata Tables.

MXCI SQL/MX Conversational Interface

MXCI, the SQL/MX conversational interface, is useful for running ad hoc queries and for qualitatively comparing the relative efficiency of various queries.

MXCI Session

You start MXCI by using the `mxci` command within the OSS environment. For example, the commands you enter and the MXCI banner might look like this:

```
$DATA06 TEMP 19> osh
/G/SYSTEM/SYSTEM: mxci
Hewlett-Packard NonStop(TM) SQL/MX Conversational Interface 2.3
(c) Copyright 2007 Hewlett-Packard Development Company, LP.
>>
```

Note. OSS is case-sensitive. You must enter the MXCI command in lower case.

During a session, MXCI prompts you to enter SQL/MX statements or MXCI commands with one of these prompts:

>>	The standard prompt. Enter any MXCI command, Report Writer command, or SQL/MX statement.
+>	The continuation prompt. Continue the MXCI command or SQL/MX statement from the previous line or enter a semicolon to end it.
..	The FC command prompt. See FC Command on page 4-30.
S>	The Report Writer select-in-progress prompt, available only in Report Writer mode. For more information, see the *Report Writer Guide*.
CS>	The MX Connectivity Services prompt, available only in MXCS mode.

You end an MXCI session with the EXIT command. See EXIT Command on page 4-29.

After exit, the MXCI returns completion status for the session as follows:

0 - All statements executed without any errors or warnings.

1 - One or more statements in the MXCI session returned warnings.

2 - One or more statements in the MXCI session returned errors.

Session Attributes

Within an MXCI session, the tasks you perform are affected by these attributes of the session:

Parameters	Parameter values set by the SET PARAM command to substitute for parameter names when a statement executes
Prepared Statements	Statements compiled for execution later in the session
Transactions	Transaction modes set by the SET TRANSACTION statement for the next transaction in the session
NAMETYPE value	Attribute value ANSI or NSK indicates whether partially-qualified object names in SQL statements executed in the session are logical names (ANSI) or Guardian physical names (NSK).
Default Names	Default logical catalog and schema names for unqualified names in preparable statements and set by the SET CATALOG and SET SCHEMA statements respectively. Default physical volume and subvolume names for unqualified names in preparable statements and set by the SET MPLOC statement.

Entering a Command

Each statement or command entered through MXCI must be terminated by a semicolon (`;`). You can include several statements or commands on the same line provided that each one is terminated by a semicolon.

Case Sensitivity

In MXCI, statements and commands can be in uppercase, lowercase, or mixed case letters. All parts of statements and commands are case-insensitive except for parts that you enclose in single- or double-quotes.

Breaking the Command Line

You can continue any statement or command over multiple lines, breaking that statement or command at any point except within a word, a numeric literal, or a multicharacter operator (for example, >=). When you break a string literal, you must use the concatenation operator (||). A semicolon terminates a statement or command that is broken over several lines.

The maximum length of a MXCI statement in NonStop SQL/MX is 4096 characters, without any new lines or embedded carriage returns.

SQL Comments

You can include comments in MXCI input lines. Comments are useful for describing a statement or command. You can also use comments to disable specific statements or commands without removing them from the source code, such as for debugging purposes.

To indicate that an MXCI line is a comment, precede the comment with two hyphens (--):

-- *comment-text*

All text between two hyphens and the end of the physical line is a comment. You can include a comment within a statement or command (but not within a literal) if you use more than one physical line to enter the statement or command.

If you include comments in an MXCI command file, MXCI prints the comments along with the commands as it executes the file.

Examples of SQL Comments

- Show comments and SQL statements from an MXCI command file—and the output from the SQL statements—as displayed by MXCI when the file is executed:

```
-- This example shows a use of the AVG function
SELECT AVG (salary) FROM persnl.employee;
(EXPR)
----------------
        49441.52

--- 1 row(s) selected.

-- This example shows a use of the DISTINCT clause.
SELECT AVG (DISTINCT SALARY)
FROM PERSNL.EMPLOYEE;
(EXPR)
----------------
        53609.89

--- 1 row(s) selected.
```

Transactions in MXCI

A transaction can be user-defined or system-defined. If you attempt to exit an MXCI session when either type of transaction is active, MXCI prompts you to specify whether to commit or roll back the work of the transaction as follows:

```
There is an active transaction.  Do you want to commit the
transaction?
                 Y to commit transaction
                 N to abort transaction
                 Any other key to resume:
```

During an MXCI session, if the input is read from a file rather than from the keyboard and a transaction is active when MXCI reaches the end of the input file, that transaction is rolled back. You must issue a COMMIT WORK or ROLLBACK WORK explicitly within the command file (after the DML statements).

Query Interruption and Termination in MXCI

In MXCI, you can interrupt and terminate a statement or command by pressing the MXCI break key. The MXCI break key can be either Ctrl-c, Ctrl-Break, or the OutsideView Break icon, depending on your interface.

After you press the MXCI break key, the statement or command terminates, and MXCI returns this message and prompts you to enter another statement or command:

```
***WARNING[15033] Break was received.

>>
```

When you use the MXCI break key to terminate a transaction, the transaction might or might not be rolled back. Execute the SHOW SESSION command to determine the status of the transaction.

Security

Access permissions on SQL/MP objects are maintained by the Guardian environment. Each object has associated security values that determine who can read, write to, execute, and purge the object (“RWEP”). These security values are used by NonStop SQL/MX to authorize access.

Access to SQL/MX objects follows the rules of the ANSI/ISO/IEC 9075:1999 SQL standard (SQL:1999). SQL:1999 uses authorization IDs to identify users during the processing of SQL statements. An SQL/MX authorization ID is either a valid Guardian user name, enclosed in double quotes or the special authorization ID, PUBLIC.

SQL:1999 specifies two special authorization IDs:

- PUBLIC - the set of all authorization IDs known to the network at the present and all future times.

- SYSTEM - the implicit grantor of privileges to the owners of objects. You cannot specify SYSTEM on any DDL statement. It is an internal ID, mentioned here only because it may be visible from a query of the metadata.

The Super ID

In SQL:1999, the creator of an object is the owner of the object. In addition, NonStop SQL/MX enables the super ID, corresponding to Guardian user-id (255,255), to act as the owner of any object on a given node.

The super ID can create objects in a schema owned by any user. However, when the super ID creates an object in a schema owned by some other user, the actual owner of that object is that user, not the super ID. In addition to creating objects, the super ID can grant or revoke privileges on objects on behalf of users who have the privilege of performing this grant or revoke action.

The super ID can perform DDL operations on any object on behalf of the object’s owner.

Guardian User ID

Each user authorized to log on to a node is identified by a Guardian user ID that consists of a group and user identification. The user ID has one of these forms:

group_number,user_number or
group_name.user_name

Guardian Super ID

Each node has one special user ID called the super ID that has Guardian group 255 and user number 255. The super ID has one of these forms:

255,255 or
SUPER.SUPER

The Super ID can act as the owner of any Guardian file on the node and any object within the domain of the SQL/MX system catalog on that node.

Security Administrator

A SQL/MX Security Administrator (SA) is a Guardian user ID designated to hold security administrator privileges by a GRANT SECURITY_ADMIN statement. Security administrator privileges allow a security administrator to GRANT and REVOKE object privileges to other users without having access to the objects themselves and to change the ownership of any SQL/MX object. The domain of a Security Administrator's privileges is all the objects within the SQL/MX system catalog within which that user is designated a Security Administrator (that is granted SECURITY_ADMIN).

Security Administrator's Group

The Security Administrator's Group (SAG) is the set of currently defined security administrators within the domain of a SQL/MX system catalog. The set might be empty.

With Grant Option

With Grant Option (WGO) is specified by SQL:1999 and refers to the right to grant a privilege to another user and also grant WGO with that privilege. All inherent privileges held by the owner of an object are held WGO. The With Grant Option must be explicitly included in a GRANT statement to convey the WGO privilege, otherwise by default, the grantee(s) of the privilege may not grant that privilege to other users.

Owner-Derived Grant

An owner-derived grant is an access privilege conveyed to a user through a privilege held WGO that can be traced back to the owner of an object. Owner-derived grants include the inherent privileges held by object owners, grants from the owner to other users, subsequent grants made by those users, and so on. The set of these grants can be visualized as hierarchies of grants rooted in each inherent privilege of the object owners (the owner-derived hierarchy of grants).

Derived Privilege

A derived privilege is a privilege held as a result of an owner-derived grant.

Derived WGO

Derived WGO refers to a privilege held as a result of an owner-derived grant With Grant Option.

Security Administrator Grant

A security administrator grant is an access privilege conveyed to a user that is distinct from an owner-derived grant and is therefore outside the owner-derived hierarchy of grants.

Data Consistency and Access Options

Access options for DML statements affect the consistency of the data that your query accesses.

For any DML statement, you specify access options by using the `FOR` *`option`* `ACCESS` clause and, for a SELECT statement, by using this same clause, you can also specify access options for individual tables referenced in the FROM clause.

The possible settings for *`option`* in a DML statement are:

READ COMMITTED	Specifies that the data accessed by the DML statement must be from committed rows.
READ UNCOMMITTED	Specifies that the data accessed by the SELECT statement need not be from committed rows. The BROWSE and READ UNCOMMITTED settings are equivalent.
SERIALIZABLE or REPEATABLE READ	Specifies that the DML statement and any concurrent process (accessing the same data) execute as if the statement and the other process had run serially rather than concurrently.
SKIP CONFLICT	Allows transactions to skip rows locked in a conflicting mode by another transaction. SKIP CONFLICT cannot be used in a SET TRANSACTION statement.
STABLE	Specifies that the row being accessed by the SELECT statement is locked while it is processed, but concurrent use of the database is allowed. STABLE is an ANSI extension.

The SQL/MX default access option for DML statements is READ COMMITTED. However, you can set your system default for access options by specifying entries in the SYSTEM_DEFAULTS table. See ISOLATION_LEVEL on page 10-56.

The implementation for REPEATABLE READ and SERIALIZABLE access options is equivalent. This entry uses SERIALIZABLE for purposes of illustration.

For related information about transactions, see Transaction Isolation Levels on page 1-23.

SQL/MP Considerations

Note. If your SQL/MP application uses the BROWSE, STABLE, and REPEATABLE keywords, NonStop SQL/MX accepts these keywords as synonyms for statement-level access options READ UNCOMMITTED, STABLE, and SERIALIZABLE (or REPEATABLE READ), respectively.

READ UNCOMMITTED

This option enables you to access locked data. READ UNCOMMITTED is not available for DML statements that modify the database. It is available only for a SELECT statement.

READ UNCOMMITTED provides the lowest level of data consistency. A SELECT statement executing with this access option is allowed to:

- Read data modified by a concurrent process (sometimes referred to as *dirty reads*)
- Read different committed values for the same item at different times or find that the item no longer exists (sometimes referred to as *nonrepeatable reads*)
- Read different sets of committed values satisfying the same predicate at different times (sometimes referred to as *phantoms*)

READ COMMITTED

This option allows you to access only committed data.

The implementation requires that a lock can be acquired on the data requested by the DML statement—but does not actually lock the data, thereby reducing lock request conflicts. If a lock cannot be granted (implying that the row contains uncommitted data), the DML statement request waits until the lock in place is released.

READ COMMITTED provides the next higher level of data consistency (compared to READ UNCOMMITTED). A statement executing with this access option does not allow dirty reads, but both nonrepeatable reads and phantoms are possible.

READ COMMITTED provides sufficient consistency for any process that does not require a repeatable read capability.

SERIALIZABLE or REPEATABLE READ

This option locks all data accessed through the DML statement and holds the locks on data in audited tables until the end of any containing transaction.

SERIALIZABLE (or REPEATABLE READ) provides the highest level of data consistency. A statement executing with this access option does not allow dirty reads, nonrepeatable reads, or phantoms.

SKIP CONFLICT

This option allows transactions to skip rows locked in a conflicting mode by another transaction. Do not use SKIP CONFLICT in a SET TRANSACTION statement. For more information on the skip conflict access method, see the *SQL/MX Queuing and Publish/Subscribe Services* manual.

STABLE

For nonaudited tables, access becomes READ COMMITTED.

This option locks all data accessed through the DML statement but releases locks on unmodified data as soon as possible, which enables concurrent use of the database. STABLE access locks modified data in audited tables until the end of the transaction. STABLE is not available for DML statements that modify the database. It is available only for a SELECT statement.

In host programs that use cursors, STABLE locks an unmodified row only when the row is in the current position and releases the lock at the next FETCH that fills the buffer.

You can control the number of rows read into this buffer with the MAX_ROWS_LOCKED_FOR_STABLE_ACCESS system default attribute. The default is one row, and the maximum number of rows depends on the size of the buffer. To increase concurrency, you can decrease this value so that more messages are used to return the same amount of data.

CLOSE *cursor-name* releases the lock from the last FETCH.

STABLE is available only with updatable cursors. If a SELECT statement cannot be completed, access becomes READ COMMITTED.

For modified rows in audited tables, STABLE access uses exclusive locks held by the TMF transaction that are released only when the entire transaction ends.

STABLE access provides sufficient consistency for any process that does not require a repeatable read capability.

Database Integrity and Locking

To protect the integrity of the database, NonStop SQL/MX provides locks on data. For example, NonStop SQL/MX locks a row when an executing process (either MXCI or a host program) accesses a row to modify it. The lock ensures that no other process simultaneously modifies the same row.

Default locking normally protects data but reduces concurrency. If your application has problems with lock contention, you might want to use options that control the characteristics of locks.

Locks have these characteristics:

- Lock Duration (short or long)
- Lock Granularity (table lock, partition lock, subset of rows, or single row)
- Lock Mode (exclusive, shared, no lock)
- Lock Holder (transaction or process)

Lock Duration

Lock duration controls how long a lock is held. You can specify lock duration for only the read portion of a statement. All write locks are held until the end of the transaction (for audited tables) or until the program releases the locks (for nonaudited tables).

You can use the LOCK TABLE statement to lock a table. How long the lock is held depends on whether the locked table is audited or nonaudited.

Lock duration is also affected by whether you choose the SERIALIZABLE access option for DML statements. This access option causes the maximum lock duration.

Lock Granularity

Lock granularity controls the number of rows affected by a single lock. The level of granularity can be a table, a partition, a subset of rows, or a single row.

You can control locks for the entire table with LOCK TABLE. Otherwise, NonStop SQL/MX determines the granularity by considering the access option you specify, the table size and definition, and the estimated percentage of rows the query will access.

NonStop SQL/MX can automatically increase the granularity of locks for a particular transaction, depending on processing requirements. This increase in granularity is referred to as lock escalation. If a process holds many row locks on the same partition of a partitioned table, NonStop SQL/MX might escalate the row locks to a partition lock. For a nonpartitioned table, a partition lock is a table lock.

Lock escalation is affected by the value of MaxLocksPerTCB. If an application acquires locks on more than 10% of this value, or a default of 500, since MaxLocksPerTCB's default is 5000 for the volume, DP2 attempts to escalate to a table lock, even though these rows may be locked only on a single partition of the table. If there are other transactions running concurrently with locks on the table, DP2 may not be able to escalate to a table lock, but it will keep trying.

You can prevent this by setting the TABLELOCK default to OFF. However, if the number of rows locked reaches the value of MaxLocksPerTCB, that transaction will not be able to obtain any more locks, and may be aborted. Depending on the operation, some updates must be backed out which can seriously affect the application's performance.

You can control the number of locks that a transaction can hold on a specific volume and lock escalation by using SCF to change MaxLocksPerTCB to a maximum of 100,000. Use this command:

```
SCF ALTER $volume, MAXLOCKSPERTCB n
```

Lock Mode

Lock mode controls access to locked data. You can specify lock mode only for rows that are read.

SHARE lock mode allows multiple users to lock and read the same data. EXCLUSIVE lock mode limits access to locked data to the lock holder and to other users who specify READ UNCOMMITTED (but not READ COMMITTED or SERIALIZABLE) access. Lock modes are the same when you choose READ COMMITTED or SERIALIZABLE access.

Lock mode is sometimes determined by NonStop SQL/MX. NonStop SQL/MX ensures that an exclusive lock is in effect for write operations and usually acquires a shared lock for operations that access data without modifying it. You choose lock mode in these instances:

- On the LOCK TABLE statement, you can choose either EXCLUSIVE or SHARE.
- On the SELECT statement, you can specify IN EXCLUSIVE MODE or IN SHARE MODE.

Lock Holder

The lock holder of an object depends on whether the object is audited or nonaudited:

- Locks on audited objects are held by the transaction in which the request to access the data was made.
- Locks on nonaudited objects are held by the process that opens the object: either MXCI or a host program.

Only the lock holder can release a lock:

- A transaction releases the locks it holds at the end of the transaction in either of these cases:
    - Locks on data read using SERIALIZABLE access
    - Locks on rows updated

- A process can hold a lock over the duration of one (or more) transactions, or the process can release the lock before the transaction completes. A process releases the locks it holds by issuing statements that affect the locks.

Stopping or abnormal termination of a process frees any locks the process holds on nonaudited tables.

Transaction Management

A transaction (a set of database changes that must be completed as a group) is the basic recoverable unit in case of a failure or transaction interruption. Transactions can be defined during an MXCI session or in a host program. The typical order of events is:

1. Transaction is started.
2. Database changes are made.
3. Transaction is committed.

If, however, the changes cannot be made or if you do not want to complete the transaction, you can abort the transaction so that the database is rolled back to its original state.

All SQL/MX tables must be audited, and although SQL/MP tables can be nonaudited, HP recommends that they be audited. Transactions are managed by the HP NonStop Transaction Management Facility (TMF). This product simplifies the task of maintaining data consistency for databases being updated by concurrent transactions. For more information on TMF, see the *Transaction Management Facility (TMF) Introduction*.

Any transaction is subject to TMF's two-hour limit on audit trails. TMF will automatically abort a query that runs longer than two hours. You can change this limit to a maximum of 5965 hours (about 8 months) or set it to zero. In that case, TMF will never perform an AUTOABORT. This limit can help protect your application from runaway queries or transactions.

In spite of the AUTOABORT setting, TMF still aborts any transaction or query that pins the oldest MAT (master audit file) if the file is pinned because of currently active transactions, and if audit information is filling 45% or more of the MAT's capacity.

Choose this setting with care. Increasing TMF limits degrades system performance and increases disk space usage for the audit trail.

If you are running a business intelligence system, base the setting on how long you expect your longest query to run. If you are running an online transaction environment, base the setting on the longest running update transaction that you plan to have. It is preferable to have short running transactions or batch updates with frequent COMMIT WORK statements.

This subsection discusses these considerations for transaction management:

- Statement Atomicity on page 1-14
- User-Defined and System-Defined Transactions on page 1-15

- Rules for DML Statements on page 1-16
- Effect of AUTOCOMMIT Option on page 1-16
- Concurrency on page 1-16
- Transaction Access Modes on page 1-23
- Transaction Isolation Levels on page 1-23

Statement Atomicity

To maintain database consistency, transactions must be controlled so that they either complete successfully or are aborted. SQL/MX Release 1.8 automatically aborted transactions if an error occurred while performing an SQL statement. SQL/MX Release 2.x by default does not automatically abort transactions following an error, in most situations.

SQL/MX Release 2.x guarantees that an individual SQL statement within a transaction either completes successfully or has no effect on the database. To retain the behavior of SQL/MX Release 1.8, use the UPD_ABORT_ON_ERROR default. For more information, see Statement Atomicity on page 10-78.

When an INSERT, UPDATE, or DELETE statement encounters an error, that transaction is not aborted, but continues. The effect of the SQL/MX statement is rolled back, so the statement has no effect on the database, but the transaction is not aborted. This functionality is provided through an internal feature called *savepoints*.

Statement atomicity occurs if these conditions are met:

- The UPD_ABORT_ON_ERROR default must be set to OFF (the default.)
- The underlying table must not have referential integrity constraints, or triggers.
- The SQL query is not:
  - A publish/subscribe query with stream access
  - A CALL statement
  - A holdable cursor
  - A SELECT statement with an embedded UPDATE or DELETE
  - A DDL statement
  - An UPDATE STATISTICS statement
- The query plan does not choose VSBB inserts or Executor Server Process (ESP) parallelism.
- The AUTOCOMMIT option must be set to ON.

If these conditions are not met, the transaction is aborted by NonStop SQL/MX if a failure occurs. This behavior occurs for all INSERT, UPDATE, or DELETE statements in SQL/MX prior to SQL/MX Release 2.x.

When NonStop SQL/MX attempts to perform an insert, update, or delete transaction against a single row, it does not use savepoints. If the operation fails, NonStop SQL/MX returns an error. Because no change was made to the database, nothing is rolled back.

When NonStop SQL/MX attempts to insert, update, or delete multiple rows, it uses savepoints and if it encounters an error during the operation it issues a warning, rolls back that statement, and continues.

For more information on the UPD_ABORT_ON_ERROR default, see Statement Atomicity on page 10-78.

If the default INSERT_VSBB is set to USER, NonStop SQL/MX will not use statement atomicity. Unless you are inserting only a few records you should not disable INSERT_VSBB to use statement atomicity because performance will be affected. Perform UPDATE STATISTICS on the tables so that row estimates are correct.

To see what rollback mode NonStop SQL/MX is choosing, you can prepare the query, then perform EXPLAIN.

```
explain options 'f' my_query;
```

The OPT column displays token upd_action_on_error: on_rollback. A value of “x” means that the transaction will be rolled back. Any other value means the transaction will be aborted. For details about the output options, see EXPLAIN Statement on page 2-208, INSERT_VSBB on page 10-75, and UPD_ABORT_ON_ERROR on page 10-78.

For more information about the differences in auto-abort behavior between NonStop SQL/MP and NonStop SQL/MX, see the *SQL/MX Comparison Guide for SQL/MP Users*.

User-Defined and System-Defined Transactions

User-Defined Transactions

Transactions you define are called *user-defined transactions*. To ensure that a sequence of statements either executes successfully or not at all, you can define one transaction consisting of these statements by using the BEGIN WORK Statement and COMMIT WORK Statement. You can abort a transaction by using the ROLLBACK WORK Statement.

System-Defined Transactions

In some cases, NonStop SQL/MX defines transactions for you. These transactions are called *system-defined transactions*. Most DML statements initiate transactions implicitly at the start of execution. See Implicit Transactions on page 2-378. However, even if a transaction is initiated implicitly, you must end a transaction explicitly with the COMMIT WORK statement or the ROLLBACK WORK statement.

Rules for DML Statements

- DML statements executing on audited tables, views of audited tables, and mixed views must be performed within a transaction, except when reading data with READ UNCOMMITED ACCESS.
- If deadlock occurs, the DML statement is canceled, but the transaction continues.

Audited and Nonaudited Tables

The TMF product works only on audited tables, so a transaction does not protect operations on nonaudited tables. The simplest approach is to make all tables audited. The AUDIT file attribute is the default when a table is created.

Nonaudited tables are not protected by transactions and follow a different locking and error handling model than audited tables. Certain situations such as DML error occurrences can lead to inconsistent data within a nonaudited table or between a nonaudited table and its indices.

Effect of AUTOCOMMIT Option

AUTOCOMMIT is an option that can be set in a SET TRANSACTION statement. It specifies whether NonStop SQL/MX will commit automatically, or roll back if an error occurs, at the end of statement execution. This option applies to any statement for which the system initiates a transaction. See SET TRANSACTION Statement on page 2-376.

If this option is set to ON, NonStop SQL/MX automatically commits any changes, or rolls back any changes, made to the database at the end of statement execution. AUTOCOMMIT is set ON by default at the start of an MXCI session.

If this option is set to OFF, the current transaction remains active until the end of the MXCI session unless you explicitly commit or roll back the transaction. The default is OFF for embedded SQL in a C or COBOL program. The default is ON for embedded SQL in a Java program.

Concurrency

Concurrency is defined by two or more processes accessing the same data at the same time. The degree of concurrency available—whether a process that requests access to data that is already being accessed is given access or placed in a wait queue—depends on the purpose of the access mode (read or update) and the isolation level.

NonStop SQL/MX provides concurrent database access for most operations and controls database access through the mechanism for locking and the mechanism for opening and closing tables. For DML operations, access and locking options affect the degree of concurrency. See Data Consistency and Access Options on page 1-8, Database Integrity and Locking on page 1-11, and SET TRANSACTION Statement on page 2-376.

These tables describe interactions between SQL/MX operations:

Table 1-1 on page 1-17 compares operations with access modes and lists DDL and Utility operations you can start while DML operations are in progress.

Table 1-1. Concurrent DDL/Utility Operation and File Access Modes

	Access Mode			
DDL Operations You Can Start	**READ UNCOMMITTED**	**READ COMMITTED**	**STABLE**	**SERIALIZABLE**
ALTER INDEX	Allowed	Allowed	Allowed	Allowed
ALTER SEQUENCE attributes	Allowed*	Allowed*	Waits	Waits
ALTER TABLE attributes	Allowed*	Allowed*	Waits	Waits

* DDL operation aborts the DML operation

Table 1-2 compares DDL and utility operations with DML operations and shows DDL operations you can start while DML operations are in progress:

Table 1-2. Concurrent DDL/Utility and DML Operations

	DML Operation in Progress			
DDL Operations You Can Start	**SELECT UNCOMMITTED**	**SELECT SHARE**	**SELECT EXCLUSIVE**	**UPDATE/ INSERT/ DELETE**
ALTER INDEX	Allowed*	Allowed	Allowed	Allowed
ALTER SEQUENCE attributes	Allowed*	Allowed	Allowed	Allowed
ALTER TABLE attributes	Allowed*	Allowed	Allowed	Allowed
ALTER TABLE other	Allowed*	Waits	Waits	Waits
CREATE INDEX with POPULATE	Allowed*	Allowed	Waits	Waits
CREATE INDEX NO POPULATE	Allowed	Allowed	Allowed	Allowed
CREATE SEQUENCE	Allowed	Allowed	Allowed	Allowed
CREATE TRIGGER subject table	Allowed	Allowed	Waits	Waits
CREATE TRIGGER referenced table	Allowed	Allowed	Allowed	Allowed
CREATE VIEW	Allowed	Allowed	Allowed	Allowed
GRANT	Allowed*	Waits	Waits	Waits
MODIFY online operations	Allowed*	Allowed**	Allowed**	Allowed**
MODIFY offline operations***	Allowed*	Allowed**	Allowed**	Waits

Table 1-2. Concurrent DDL/Utility and DML Operations

	DML Operation in Progress			
DDL Operations You Can Start	**SELECT UNCOMMITTED**	**SELECT SHARE**	**SELECT EXCLUSIVE**	**UPDATE/ INSERT/ DELETE**
POPULATE INDEX	Allowed*	Allowed**	Allowed**	Waits
REVOKE	Allowed*	Allowed	Waits	Waits
UPDATE STATISTICS	Allowed	Allowed	Allowed	Allowed**

* DDL operation aborts the DML operation
** Allowed except during commit phase
*** There are some exceptions. Dropping a partition from a hash partitioned table or index requires exclusive access.

Table 1-3 compares DML operations you can start when DDL operations are in progress:

Table 1-3. Concurrent DML and DDL Operations (page 1 of 2)

	DML Operations You Can Start			
DDL Operations in Progress	**SELECT UNCOMMITTED**	**SELECT SHARE**	**SELECT EXCLUSIVE**	**UPDATE/ INSERT DELETE**
ALTER INDEX	Allowed*	Allowed	Allowed	Allowed
ALTER SEQUENCE attributes	Allowed*	Allowed	Allowed	Allowed
ALTER TABLE attributes	Allowed*	Allowed	Allowed	Allowed
ALTER TABLE other	Allowed*	Waits	Waits	Waits
CREATE INDEX with POPULATE	Allowed*	Allowed	Waits	Waits
CREATE INDEX NO POPULATE	Allowed	Allowed	Allowed	Allowed
CREATE SEQUENCE	Allowed	Allowed	Allowed	Allowed
CREATE TRIGGER subject table	Allowed	Allowed	Waits	Waits
CREATE TRIGGER referenced table	Allowed	Allowed	Allowed	Allowed
CREATE VIEW	Allowed	Allowed	Allowed	Allowed
GRANT	Allowed*	Waits	Waits	Waits
MODIFY online operations	Allowed*	Allowed	Waits	Waits
MODIFY offline operations***	Allowed*	Allowed**	Allowed**	Waits
POPULATE INDEX	Allowed*	Allowed**	Allowed**	Waits

Table 1-3. Concurrent DML and DDL Operations (page 2 of 2)

		DML Operations You Can Start		
REVOKE	Allowed*	Allowed	Waits	Waits
UPDATE STATISTICS	Allowed	Allowed	Allowed	Allowed**

* DDL operation aborts the DML operation
** Allowed except during commit phase
*** There are some exceptions. Dropping a partition from a hash partitioned table or index requires exclusive access.

Table 1-4 describes the effect of various DDL and utility operations on table timestamps:

Table 1-4. Operations Effect on Table Timestamps (page 1 of 2)

Alter Operation	Timestamp Updated
ALTER INDEX	No
ALTER SQL/MP ALIAS	No*
ALTER SEQUENCE	No
ALTER TABLE	Yes, if you add columns or add or drop constraints No, if you change attributes
ALTER TRIGGER	No
BACKUP	No
CREATE CATALOG	No
CREATE INDEX	Yes, if populated
CREATE PROCEDURE	No
CREATE SCHEMA	No
CREATE SQLMP ALIAS	No
CREATE SEQUENCE	No
CREATE TABLE	No
CREATE TRIGGER	Yes, of the table on which the trigger is defined
CREATE VIEW	No
DUP	No
FIXUP	Yes
GRANT	No
IMPORT	Yes, if using fast load technique
INFO	No
MODIFY, all forms	Yes
mxexportddl	No
MXGNAMES	No
POPULATE INDEX	Yes

Table 1-4. Operations Effect on Table Timestamps (page 2 of 2)

Alter Operation	Timestamp Updated
PURGEDATA	Yes
RESTORE	Yes**
REVOKE	No
UPDATE STATISTICS	No
VERIFY	No

* Manual recompilation might be required.
** If you restore an entire table (including metadata), all timestamps are updated. The table needs to be dropped and re-created.

If you restore with PARTONLY (only one or more partitions are restored) or if you restore the entire partition (that is, the partition did not exist in the target table before restore), the redefinition time stamp of the table is updated.

If only data is restored (the partition existed in the target table before the restore), the last open timestamp of partition data fork is updated, and the data modification timestamp of the partition data fork is updated.

Table 1-5 lists concurrency limits on utilities.

Table 1-5. Concurrency Limits on Utility Operations (page 1 of 3)

Utility	DML operations	Other utilities	DDL operations
Utilities that only read metadata information: EXPORTDDL INFO MXGNAMES SHOWDDL SHOWLABEL VERIFY	All DML operations (SELECT, UPDATE, DELETE, INSERT) can be performed concurrently.	Any utility in this category can be performed concurrently.	Not recommended.
Utilities that read both metadata and user data: BACKUP FASTCOPY (source table only) DUP (source table only)	Only SELECT is allowed.	Utilities that read metadata can be performed only concurrently.	Not recommended.

Table 1-5. Concurrency Limits on Utility Operations (page 2 of 3)

Utility	DML operations	Other utilities	DDL operations
Utilities that read metadata and update user data: IMPORT (not using fast load)	All DML operations (SELECT, UPDATE, DELETE, INSERT) can be performed concurrently. If there are too many locks on the partition, DP2 escalates to a table lock which prevents concurrent DML operations.	Utilities that read metadata can only be performed concurrently. Parallel imports on the same table are allowed.	Not recommended.
Utilities that update metadata and read and write user data offline: MODIFY without SHARED access* POPULATE INDEX	Only SELECT is allowed.**	Utilities that read metadata can only be performed concurrently.	Not allowed.
Utilities that update metadata and read and potentially write user data online: MODIFY with SHARED access UPDATE STATISTICS	All DML operations are allowed.	Utilities that read metadata can only be performed concurrently.***	Not allowed.
Utilities that update data and potentially change metadata: FIXUP FASTCOPY (target table only) DUP (target table only) IMPORT using fast load MODIFY when dropping a hash partition PURGEDATA RESTORE	Not allowed.	Concurrent operations are not allowed.	Not allowed.
Utilities that read and update metadata, but do not change user data: UPGRADE DOWNGRADE	All DML operations.	Utilities that only read metadata.	Not allowed.

Table 1-5. Concurrency Limits on Utility Operations (page 3 of 3)

Utility	DML operations	Other utilities	DDL operations

* There are some exceptions. Dropping a partition from a hash partitioned table or index requires exclusive access.
** The last phase of these operations requires exclusive access to the table or index, which prevents even SELECT operations.
*** The last phase of the MODIFY operation requires exclusive access to the table or index, which prevents all DDL and DML operations.

Transaction Access Modes

A transaction has an access mode that is either READ ONLY or READ WRITE. You can set the access mode of a transaction by using a SET TRANSACTION statement. See SET TRANSACTION Statement on page 2-376.

READ ONLY

If a transaction is executing with the READ ONLY access mode, statements within the transaction can read, but cannot insert, delete, or update, data in tables. This restriction means that among the DML statements, only the SELECT statement can execute within that transaction.

If the transaction isolation level is READ UNCOMMITTED, the default access mode is READ ONLY. Further, for READ UNCOMMITTED, you can specify only READ ONLY explicitly by using the SET TRANSACTION statement.

READ WRITE

If a transaction is executing with the READ WRITE access mode, statements within the transaction can read, insert, delete, or update data in tables. Therefore, any DML statement can execute within that transaction.

If the transaction isolation level is not READ UNCOMMITTED, the default access mode is READ WRITE. However, you can specify READ ONLY explicitly by using the SET TRANSACTION statement.

Transaction Isolation Levels

A transaction has an isolation level that is either READ UNCOMMITTED, READ COMMITTED, or SERIALIZABLE or REPEATABLE READ. The SQL/MX implementation for REPEATABLE READ and SERIALIZABLE is equivalent. SERIALIZABLE is used for purposes of illustration.

You can set the isolation level of a transaction explicitly by using a SET TRANSACTION statement. See SET TRANSACTION Statement on page 2-376.

You can set your system default for the transaction isolation level by specifying the ISOLATION_LEVEL entry in the SYSTEM_DEFAULTS table. The default isolation level of a transaction is determined according to the rules specified in ISOLATION_LEVEL on page 10-56.

READ UNCOMMITTED

This isolation level allows your transaction to access locked data. You cannot use READ UNCOMMITTED for transactions that modify the database.

READ UNCOMMITTED provides the lowest level of data consistency. A transaction executing with this isolation level is allowed to:

- Read data modified by a concurrent transaction (sometimes referred to as *dirty reads*)
- Read different committed values for the same item at different times or find that the item no longer exists (sometimes referred to as *nonrepeatable reads*)
- Read different sets of committed values satisfying the same predicate at different times (sometimes referred to as *phantoms*)

READ COMMITTED

This option allows your transaction to access only committed data.

The implementation requires that a lock can be acquired on the requested data—but does not actually lock the data, thereby reducing lock request conflicts. If a lock cannot be granted (implying that the row contains uncommitted data), the transaction request waits until the lock in place is released.

READ COMMITTED provides the next level of data consistency. A transaction executing with this isolation level does not allow dirty reads, but both nonrepeatable reads and phantoms are possible.

READ COMMITTED provides sufficient consistency for any transaction that does not require a repeatable-read capability.

SERIALIZABLE or REPEATABLE READ

This option locks all data accessed through the transaction and holds the locks on data in audited tables until the transaction ends.

SERIALIZABLE (or REPEATABLE READ) provides the highest level of data consistency. A transaction executing with this isolation level does not allow dirty reads, non-repeatable reads, or phantoms.

For audited tables (SQL/MX tables are audited), SERIALIZABLE uses shared locks for unmodified rows and exclusive locks for modified rows—but all locks are held by the transaction and not released until the transaction ends. SERIALIZABLE prevents other users from inserting or modifying (including delete) rows in the range of rows (key range if using unique primary key or all rows if using non-unique column) examined by the transaction.

Non-Unique Key Considerations for SERIALIZABLE or REPEATABLE READ

If the SELECT statement uses a unique column (primary key), SQL/MX locks only the rows specified in the unique key range. If the SELECT statement uses a non-unique column, SQL/MX locks all the rows (whole table) to guarantee REPEATABLE READ access. For information on locks, see Database Integrity and Locking on page 1-11.

Partition Management

You can create SQL/MX tables with multiple physical files, or partitions. Use the CREATE TABLE Statement on page 2-107 and the CREATE INDEX Statement on page 2-80 to create tables and indexes that include partitions. Use the MODIFY Utility to partition tables after they have been created.

For more information, see Partitions on page 6-83 for an overview of partitions in SQL/MX and SQL/MP files. For more information about managing partitioned files, see the *SQL/MX Installation and Management Guide*.

Internationalization

Users need to be able to display data in formats appropriate to their locale and language—in English or other Roman-character formats, in Japanese Kanji or Korean or Chinese characters. You can select from one single-character or three double-byte character sets. For more information about character sets in addition to restrictions on the use of character sets, see Character Sets on page 6-4.

Using NonStop SQL/MX to Access SQL/MP Databases

NonStop SQL/MX allows applications to use the SQL/MX engine to access SQL/MP databases. SQL/MP tables, views, indexes, and catalogs are accessed by using SQL/MX DML statements. For more information on the SQL/MP language, see the *SQL/MP Reference Manual*.

Mixing embedded SQL calls to NonStop SQL/MP and NonStop SQL/MX from the same application process is not supported.

NonStop SQL/MX provides support for nonstandard SQL/MP features so that you can develop applications that use these databases. However, when you use NonStop SQL/MX to access an SQL/MP database, you should be aware of some restrictions involving SQL/MP features that do not directly map to NonStop SQL/MX.

The areas of support and the restrictions on access are:

Naming Objects on page 1-26	To refer to SQL/MP database objects, use either four-part Guardian Physical Names, three-part Logical Names, or DEFINE Names. To accommodate the use of ANSI names, you must create Alias Mappings from ANSI names to NSK names.
Delimiting Reserved Words in Guardian Names on page 1-27	If a column or table name contains an SQL/MX reserved word, you must delimit the reserved word in double quotes (").
Selecting or Changing Data on page 1-28	You can select or change DATETIME Data, INTERVAL Data, and NCHAR Data with some restrictions.

Accessing Views on page 1-32	You can access both protection and shorthand views with the same security as within NonStop SQL/MP.
Access Options on page 1-32	You can use the SQL/MP access options as synonyms for SQL/MX access options with some restrictions.
SQL/MP Stored Text on page 1-32	You cannot access or manipulate SQL/MP tables or views that have been defined in specific ways. There are restrictions on specific types of SQL/MP stored text.
SQL/MP File Organizations on page 1-33	You cannot access SQL/MP tables that have specific file organizations
Collations on page 1-33	You cannot access any SQL/MP tables defined with collations other than those tables defined with the default collation. You cannot include the SQL/MP COLLATE option in a GROUP BY clause or an ORDER BY clause when selecting from an SQL/MP table.

Naming Objects

Refer to SQL/MP database objects through MXCI or through applications by using either physical names, logical names, or DEFINE names, as described next. For more information, see Database Object Names on page 6-13, Object Naming on page 10-60, or DEFINEs on page 6-38.

Physical Names

NonStop SQL/MP uses Guardian names as names for SQL tables, views, indexes, partitions, collations, and program modules. A portion of the Guardian name (the subvolume name) is used as an SQL/MP catalog name.

To provide flexibility, NonStop SQL/MX provides support for Guardian four-part object names of the form:

`[\`*`node`*`.][[$`*`volume`*`.]`*`subvol`*`.]`*`filename`*

In this four-part name, `\`*`node`* is the name of a node on an HP NonStop system, `$`*`volume`* is the name of a disk volume, *`subvol`* is the name of a subvolume, and *`filename`* is the name of a Guardian disk file or the name of an SQL/MP table, view, index, partition, collation, or program module.

For more information about Guardian name resolution, see Attribute Value NSK for Guardian Names and Guardian Name Resolution on page 10-62.

Logical Names

To move toward full ANSI SQL:1999 compliance, NonStop SQL/MX provides support for three-part logical object names of the form:

`[[`*`catalog.`*`]`*`schema.`*`]`*`name`*

In this three-part name, *catalog* is the first part of the name, *schema* is the second part of the name, and *table* is the third part of the name. See Catalogs on page 6-3 and Pseudocolumns on page 6-105.

For more information about logical name resolution, see Attribute Value ANSI for Logical Names on page 10-62.

Alias Mappings

To permit the use of logical names, a user table named OBJECTS stores alias names. The MP_PARTITIONS table stores mappings from logical object names to physical Guardian locations. See OBJECTS Table on page 10-22 and MP_PARTITIONS Table on page 10-22.

To create the necessary mappings from logical to physical names, use the CREATE SQLMP ALIAS statement:

```
CREATE SQLMP ALIAS catalog.schema.table
                  [\node.]$volume.subvol.filename
```

For the complete syntax and semantics, see CREATE SQLMP ALIAS Statement on page 2-104.

To use ANSI names with the DDL Statements for the Sample Database on page D-3, you must create an alias for each table that has been created.

For example, suppose that you have created the EMPLOYEE table with the physical Guardian name `$samdb.persnl.employee`. To specify the logical name `samdbcat.persnl.employee` for the employee table, enter:

```
CREATE SQLMP ALIAS samdbcat.persnl.employee
                   $samdb.persnl.employee;
```

DEFINE Names

NonStop SQL/MX supports the use of DEFINE names as logical names for tables, views, or partitions in DML statements. When NonStop SQL/MX compiles such statements, it replaces the DEFINE name (for example, =CUSTOMERS) in the statement with the associated Guardian physical name. DEFINE names can be created within MXCI or can be inherited from the TACL process or the OSS shell.

For more information about DEFINEs, see DEFINEs on page 6-38.

Delimiting Reserved Words in Guardian Names

In NonStop SQL/MP, you can use reserved words in Guardian names that identify tables, views, partitions, and collations. NonStop SQL/MX has many more reserved words than NonStop SQL/MP. If an SQL/MX reserved word occurs as part of a Guardian name, you must delimit it by enclosing it in double quotes—that is, it must be a delimited identifier. See Using SQL/MX Reserved Words in SQL/MP Names on page 6-57.

For example, suppose that the location of the OBJECTS table is `\nsk.$system.SQL`. To determine the physical name associated with a given logical SQL/MX object name, you can query the OBJECTS table:

```
SELECT guardian_name
FROM \nsk.$system."SQL".objects
WHERE logical_name = 'samdbcat.persnl.employee';
```

In this example, `"SQL"` is written as a delimited identifier because SQL is a reserved word in NonStop SQL/MX.

Selecting or Changing Data

To select or change SQL/MP data that does not directly map to SQL/MX data types and literals, you can use special extensions of NonStop SQL/MX with some restrictions.

DATETIME Data

The SQL/MP DATETIME data type is specified:

DATETIME [*start-field* TO] *end-field*

The *start-field* and *end-field* specify a range of logically contiguous fields:

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
FRACTION [(precision)]
```

The *start-field* must precede the *end-field*. The FRACTION field can include the *precision* option only if the FRACTION field is the *end-field*.

Standard DATETIME Data Types

Certain DATETIME data types map directly to the ANSI standard types—DATE, TIME, and TIMESTAMP. You can retrieve the same value as stored in an SQL/MP column with these data types or store a value with a standard type into these SQL/MP columns without truncation or extension.

This ANSI standard data type:	is equivalent to this SQL/MP data type:
DATE	DATETIME YEAR TO DAY
TIME	DATETIME HOUR TO SECOND
TIMESTAMP	DATETIME YEAR TO SECOND

For more information, see:

- SQL/MP Considerations for Datetime Data Types Not Equivalent to DATE, TIME, TIMESTAMP on page 6-27
- SQL/MP Considerations for Datetime Data Types Equivalent to DATE, TIME, TIMESTAMP on page 6-29

Truncation and Extension

If you attempt to insert a larger DATETIME value into a smaller DATETIME column, NonStop SQL/MX implicitly truncates the value only on the fractional part. In all other cases, NonStop SQL/MX returns an error. If you attempt to insert a smaller DATETIME value into a larger DATETIME column, NonStop SQL/MX returns an error.

When you are storing values in a DATETIME column, you must explicitly cast the DATETIME value in question to the desired DATETIME data type to ensure compatibility. If extension occurs on the more significant end of a value, the values for the missing fields are drawn from the fields of CURRENT_TIMESTAMP. If extension occurs on the less significant end, the values are the minimum field values.

When you are comparing datetime data with different start and end fields in a WHERE clause, you must also specify an explicit CAST to ensure compatibility.

See Casting DATETIME Data for Compatibility on page 6-30.

Using Datetime Functions

You can use SQL/MX datetime functions to select individual fields from a DATETIME column in an SQL/MP table.

See Using SQL/MX Datetime Functions on DATETIME Data on page 6-30.

Selecting Any DATETIME Column

You can select data from any DATETIME column except those consisting of FRACTION only.

If you attempt to select data from a FRACTION-only column, the value is returned as the CHAR data type consisting of a string of `'#'` characters with the same display length as the length of the column.

See Selecting DATETIME Columns in SQL/MP Tables on page 6-28.

Inserting or Updating Any DATETIME Column

NonStop SQL/MX supports inserting into or updating any columns with the DATETIME data type in SQL/MP tables except those consisting of FRACTION only. Use a special SQL/MX DATETIME literal to insert into or update a DATETIME column. The literal is specified:

```
DATETIME 'datetime' [start-field TO] end-field
```

See Inserting Into or Updating Any SQL/MP DATETIME Column on page 6-68.

INTERVAL Data

SQL/MP INTERVAL values represent durations of time in year-month units (years and months), in day-time units (days, hours, minutes, seconds, and fractions of a second), or in subsets of those units.

Year-Month Interval

Specify a year-month duration:

```
INTERVAL start-ym [(digits)] [TO end-ym]
```

The *start-ym* and *end-ym* specify a range of logically contiguous fields:

```
YEAR
MONTH
```

Day-Time Interval

Specify a day-time duration:

```
INTERVAL start-dt [(digits)] [TO end-dt]
```

The *start-dt* and *end-dt* specify a range of logically contiguous fields:

```
DAY
HOUR
MINUTE
SECOND
FRACTION [(precision)]
```

The *start-dt* must precede the *end-dt*. The FRACTION field can include the *precision* option only if the FRACTION field is the *end-dt*.

Selecting Any INTERVAL Column

You can select data from any SQL/MP INTERVAL columns with a start field of YEAR through SECOND. All SQL/MP INTERVAL data types that have a start field of YEAR through SECOND are directly compatible with their corresponding SQL/MX INTERVAL data types.

If you attempt to select data from a FRACTION-only column, the value is returned as the CHAR data type consisting of a string of `'#'` characters with the same display length as the length of the column.

See Selecting INTERVAL Columns in SQL/MP Tables on page 6-33.

Inserting or Updating Any INTERVAL Column

NonStop SQL/MX supports inserting into or updating any columns with the INTERVAL data type in SQL/MP tables except those consisting of FRACTION only. Use an INTERVAL literal to insert into or update an INTERVAL column in the usual way. The literal is specified:

```
[-]INTERVAL [-]{'year-month'|'day:time'} interval-qualifier
```

For the complete syntax of interval literals, see Interval Literals on page 6-71. See Inserting Into or Updating Any SQL/MP INTERVAL Column on page 6-73.

NCHAR Data

From SQL/MX Release 2.x, you can select character data from NCHAR columns in SQL/MP and SQL/MX tables. You can insert into or update NCHAR columns in an SQL/MP table only when the character data being written to the table contains an even number of bytes. A string literal you use this way can be specified:

```
N'string'
```

N associates the default character set with the string literal. The default character set is the NATIONAL_CHARSET attribute you specify when you install NonStop SQL/MX. For more information about setting the NCHAR default, see Character Sets on page 6-4.

For SQL/MX Release 2.x, LIKE predicates and character string functions that refer to double byte-encoded characters in NCHAR columns of SQL/MP and SQL/MX tables always provide the correct results. Character string functions include INSERT, LEFT, LOCATE, LPAD, LTRIM, POSITION, REPLACE, RIGHT, RPAD, RTRIM, SUBSTRING, and TRIM.

Because SQL/MX Release 2.x compares and sorts all character data, including double byte-encoded characters, on the character boundary instead of the byte boundary, ORDER BY and GROUP BY also return the correct results. NonStop SQL/MX uses a binary collation, so characters are always compared and sorted on the basis of their character value, not their byte length. If the character values of compared characters are the same, a match occurs.

Accessing Views

The FOR PROTECTION clause of the SQL/MP CREATE VIEW statement specifies a protection view. If you omit this clause, the view is a shorthand view.

NonStop SQL/MX provides support for the access of SQL/MP protection views. A protection view is derived from a single table and has associated security values that determine who can read, write to, execute, and purge the view. Security specifically defined on the view overrides the security on the underlying table.

NonStop SQL/MX also supports the read-only access of SQL/MP shorthand views. A shorthand view is derived from one or more tables or other views and inherits the security of the underlying table or tables.

Access Options

If your SQL/MP application uses the BROWSE, STABLE, and REPEATABLE keywords, NonStop SQL/MX accepts these keywords as synonyms for statement-level access options READ UNCOMMITTED, STABLE, and SERIALIZABLE (or REPEATABLE READ), respectively.

SQL/MP Stored Text

You cannot access or manipulate SQL/MP tables or views that have been defined in specific ways. There are restrictions on specific types of SQL/MP stored text, which is SQL text that NonStop SQL/MX retrieves from the SQL/MP catalog while processing SQL/MX text. SQL/MP stored text includes views, constraints, column defaults, first keys, clustering keys, and partitioning keys.

In NonStop SQL/MX, these types of SQL/MP stored text are disallowed:

- Views, constraints, column defaults, and first keys cannot contain:
    - UNITS function
    - DATETIME string portions with nonstandard formatting
    - FRACTION-only DATETIME or INTERVAL literals
    - Interval literals with negative signs inside quotation mark delimiters (for example, `INTERVAL '-5' DAY`)
    - Identifiers named after words that are reserved in SQL/MP stored text (see Appendix B, Reserved Words)

  Views, however, can contain a select of a FRACTION-only column.
- Clustering or partitioning keys cannot contain:
    - FRACTION-only DATETIME or INTERVAL columns
    - Interval literals with negative signs inside quotation mark delimiters

- Identifiers named after words that are reserved in SQL/MP stored text (see Appendix B, Reserved Words.)

NonStop SQL/MX supports SQL/MP double-quoted string literals, which are treated correctly as strings and not as SQL/MX delimited identifiers, in SQL/MP stored text.

NonStop SQL/MX supports SQL/MP character string literals that contain a space between the character set qualifier and character string literal, such as `_KANJI 'abcd'`, in SQL/MP stored text. NonStop SQL/MX does not allow a space after the character set qualifier in SQL/MX text. For example, you must specify `_KANJI'abcd'`in SQL/MX text. See Character String Literals on page 6-64.

NonStop SQL/MX supports equivalent syntax for the UNITS operator. See Operations Equivalent to UNITS on page 6-31.

SQL/MP File Organizations

An SQL/MP table can have one of three physical file organizations: key-sequenced, entry-sequenced, or relative. You can access these type of SQL/MP files through NonStop SQL/MX:

- Key-sequenced tables with or without partitions
- Entry-sequenced tables that are not partitioned

You cannot access these type of SQL/MP files through NonStop SQL/MX:

- Entry-sequenced tables that are partitioned
- Relative tables

For more information about SQL/MP file organizations, see the *SQL/MP Reference Manual*.

Collations

In SQL/MP tables, character columns can be sequenced by specifying a collation in the COLLATE clause of a column data type definition in a CREATE TABLE statement. In NonStop SQL/MP, you create a collation with the CREATE COLLATION statement. If you do not specify a COLLATE clause, SQL/MP character columns are sequenced by the binary value of the characters in the column.

For SQL/MX Release 2.x, you cannot access any SQL/MP tables defined with collations other than those tables defined with the default collation (consisting of the binary value of characters in the column). Further, you cannot include the SQL/MP COLLATE option in a GROUP BY clause or an ORDER BY clause when selecting from an SQL/MP table.

For more information about collations, see the CREATE COLLATION Statement, Collation Definitions, Data Types, and the COLLATE clause of the CREATE TABLE Statement in the *SQL/MP Reference Manual*.

ANSI Compliance and SQL/MX Extensions

NonStop SQL/MX complies most closely with Entry Level SQL as described in ANSI X3.135-1992 and ISO/IEC 9075:1992. NonStop SQL/MX also includes some features from Intermediate and Full Level ANSI/ISO SQL in addition to special SQL/MX extensions to the SQL language.

Statements and SQL elements in this manual are ANSI compliant unless specified as SQL/MX extensions. For details about NonStop SQL/MX's conformance with SQL:1999 standards, see Appendix E, Standard SQL and SQL/MX.

Default Settings for ANSI Compliance

To establish an ANSI-compliant database, set these default attributes as follows:

ISOLATION_LEVEL	'serializable'
NAMETYPE	'ansi'
NOT_NULL_CONSTRAINT_DROPPABLE_OPTION	'on'
PRIMARY_KEY_CONSTRAINT_DROPPABLE_OPTION	'on'
READONLY_CURSOR	'false'
REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT	'on'

To set these default attributes, use the CONTROL QUERY DEFAULT Statement on page 2-60. For more information on these default attributes, see System Defaults Table on page 10-37.

ANSI-Compliant Statements

These statements are ANSI compliant, but some might contain SQL/MX extensions:

- ALLOCATE CURSOR statement
- ALLOCATE DESCRIPTOR statement
- ALTER TABLE
- BEGIN DECLARE SECTION declaration
- CALL statement
- CLOSE statement
- COMMIT WORK statement
- CREATE PROCEDURE statement
- CREATE SCHEMA statement
- CREATE TABLE statement
- CREATE TRIGGER statement
- CREATE VIEW statement
- DEALLOCATE DESCRIPTOR statement
- DEALLOCATE PREPARE statement
- DECLARE CURSOR declaration
- DELETE statement
- DESCRIBE statement

- DROP PROCEDURE statement
- DROP SCHEMA statement
- DROP TABLE statement
- DROP TRIGGER statement
- DROP VIEW statement
- END DECLARE SECTION declaration
- EXEC SQL directive
- EXECUTE statement
- EXECUTE IMMEDIATE statement
- FETCH statement
- GET DESCRIPTOR statement
- GET DIAGNOSTICS statement
- GRANT statement
- INSERT statement
- OPEN statement
- PREPARE statement
- REVOKE statement
- ROLLBACK WORK statement
- SELECT statement
- SET statement
- SET CATALOG statement
- SET DESCRIPTOR statement
- SET SCHEMA statement
- SET TRANSACTION statement
- TABLE statement
- UPDATE statement
- VALUES statement
- WHENEVER declaration

Statements That Are SQL/MX Extensions

These statements are SQL/MX extensions to the ANSI standard. This list does not include MXCI commands, all of which are SQL/MX extensions.

- ALTER INDEX statement
- ALTER SEQUENCE statement
- ALTER SQLMP ALIAS statement
- ALTER TRIGGER
- BEGIN WORK statement
- Compound (BEGIN...END) statement
- CONTROL QUERY DEFAULT statement
- CONTROL QUERY SHAPE statement
- CONTROL TABLE statement
- CREATE CATALOG statement
- CREATE INDEX statement
- CREATE SEQUENCE statement
- CREATE SQLMP ALIAS statement
- DECLARE CATALOG statement

- DECLARE MPLOC statement
- DECLARE NAMETYPE statement
- DECLARE SCHEMA statement
- DROP CATALOG statement
- DROP INDEX statement
- DROP SEQUENCE statement
- DROP SQL statement
- DROP SQLMP ALIAS statement
- IF statement
- GET ALL SECURITY_ADMINS statement
- GIVE CATALOG statement
- GIVE Object statement
- GRANT CREATE CATALOG statement
- GRANT CREATE SCHEMA statement
- GRANT EXECUTE statement
- GRANT SECURITY_ADMIN statement
- INITIALIZE SQL statement
- INVOKE directive
- LOCK TABLE statement
- MODULE directive
- REGISTER CATALOG command
- REVOKE CREATE CATALOG statement
- REVOKE CREATE SCHEMA statement
- REVOKE EXECUTE statement
- REVOKE SECURITY_ADMIN statement
- SAMPLE clause
- SEQUENCE BY clause
- SET (assignment) statement
- SET MPLOC statement
- SET NAMETYPE statement
- SET TABLE TIMEOUT statement
- SIGNAL SQLSTATE statement
- TRANSPOSE clause
- UNLOCK TABLE statement
- UNREGISTER CATALOG command
- UPDATE STATISTICS statement

ANSI-Compliant Functions

These functions are ANSI compliant, but some might contain SQL/MX extensions:

- AVG function
- CASE expression
- CAST expression
- CHAR_LENGTH
- COUNT Function
- CURRENT
- CURRENT_DATE

- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- EXTRACT
- LOWER
- MAX
- MIN
- OCTET_LENGTH
- POSITION
- SESSION_USER
- SUBSTRING
- SUM
- TRIM
- UPPER
- USER

All other functions are SQL/MX extensions.

SQL/MX Error Messages

NonStop SQL/MX reports error messages and exception conditions within the SQL/MX conversational interface, MXCI, and in the standard output of embedded SQL programs. When an error condition occurs, NonStop SQL/MX returns a message number and a brief description of the condition. For example, NonStop SQL/MX might display this error message in MXCI:

```
*** ERROR[1000]  A syntax error occurred.
```

The message number is the SQLCODE value (without the sign). In this example, the SQLCODE value is 1000.

In MXCI, you can display the text associated with a message number (or SQLCODE value) by using the ERROR command. See ERROR Command on page 4-27. The ERROR command returns this information:

```
*** SQLSTATE (Err): 42000 SQLSTATE (Warn): 01500
*** ERROR[1000] A syntax error occurred.
```

The SQLCODE value has corresponding ANSI SQL:1999 SQLSTATE error and warning values. In this example, 42000 (error) and 01500 (warning) are the SQLSTATE values.

To view detailed cause, effect, and recovery information for ERROR[1000] and other errors, see the *SQL/MX Messages Manual*.

For more information on how to access exception conditions within embedded SQL programs, see the *SQL/MX Programming Guide for C and COBOL*.

2 SQL/MX Statements

This section describes the syntax and semantics of NonStop SQL/MX statements that you can run in MXCI or embed in programs written in C, C++, COBOL, or Java. For more information on which SQL/MX statements you can embed in a particular language, see the *SQL/MX Programming Manual for C and COBOL*.

Categories

The statements are categorized according to their functionality:

Data Definition Language (DDL) Statements

Use these DDL statements to define, delete, or modify the definition of an SQL/MX catalog, schema, or object, or the authorization to use an object.

ALTER INDEX Statement on page 2-11	Changes file attributes of an index and renames the index.
ALTER SEQUENCE Statement on page 2-13	Maps an existing alias to a different SQL/MP table.
ALTER TABLE Statement on page 2-19	Adds a constraint or column to a table, drops existing constraints, changes file attributes of a table, or renames a table.
ALTER TRIGGER Statement on page 2-48	Alters trigger status.
ALTER SEQUENCE Statement on page 2-13	Alters sequence generator attributes.

DROP TRIGGER Statement on page 2-192	Drops a trigger.
DROP VIEW Statement on page 2-193	Drops a view.
EXPLAIN Statement on page 2-208	Generates and displays the result of the EXPLAIN function.
GIVE CATALOG Statement on page 2-236	Transfers the ownership of catalog from one Guardian user to another.
GIVE Object Statement on page 2-237	Transfers the ownership of the object from one Guardian user to another.
GIVE SCHEMA Operation on page 2-239	Transfers the ownership of a schema, and optionally the ownership of objects within the schema owned by the schema owner from one user to another.
GRANT Statement on page 2-240	Grants access privileges to users for a table, view, sequence generator or stored procedure.
GRANT CREATE CATALOG Statement on page 2-244	Grants privileges to create a catalog for specified users.
GRANT CREATE SCHEMA Statement on page 2-245	Grants privileges to create a schema on a specified catalog to specified users.
GRANT EXECUTE Statement on page 2-246	Grants access privileges for a procedure to specified users.
GRANT SECURITY_ADMIN Statement on page 2-249	Grants security administration privileges to designated users.
INITIALIZE SQL Statement on page 2-251	Prepares a local node to run NonStop SQL/MX. Creates SQL/MX user metadata (UMD) tables, system metadata, and NonStop MXCS metadata tables.
REVOKE Statement on page 2-317	Revokes access privileges for a table, view, or sequence generator.
REVOKE CREATE CATALOG Statement on page 2-320	Revokes the privilege to create a catalog from specified users.
REVOKE CREATE SCHEMA Statement on page 2-321	Revokes the privilege to create a schema on a specified catalog to specified users.
REVOKE EXECUTE Statement on page 2-323	Revokes access privileges for a procedure to specified users.

REVOKE SECURITY_ADMIN Statement on page 2-326	Revokes security administration privileges from designated users.
SET Statement on page 2-365	Controls the action of a BEFORE trigger.
SIGNAL SQLSTATE Statement on page 2-381	Enables a trigger execution to raise an exception that causes both the triggered and triggering statements to fail.

Data Manipulation Language (DML) Statements

Use these DML statements to delete, insert, select, or update rows in one or more tables:

DELETE Statement on page 2-162	Deletes rows from a table or view.
INSERT Statement on page 2-252	Inserts data into tables and views.
SELECT Statement on page 2-330	Retrieves data from tables and views.
SELECT ROW COUNT Statement on page 2-363	Retrieves the count of rows from the SQL/MX table.
UPDATE Statement on page 2-385	Updates values in columns of a table or view.

For more information about DELETE, INSERT, SELECT, and UPDATE statement, see individual entries for these statements.

Transaction Control Statements

Use these statements to specify user-defined transactions and to set attributes for the next transaction:.

BEGIN WORK Statement on page 2-52	Starts a transaction.
COMMIT WORK Statement on page 2-57	Commits changes made during a transaction and ends the transaction.
ROLLBACK WORK Statement on page 2-328	Undoes changes made during a transaction and ends the transaction.
SET TRANSACTION Statement on page 2-376	Sets attributes for the next SQL transaction—the isolation level, access mode, size of the diagnostics area, and whether to automatically commit database changes.

Prepared SQL Statements

Use these statements to compile an SQL statement and then execute the statement any number of times within the current session:

EXECUTE Statement on page 2-201	Executes an SQL statement previously compiled by the PREPARE statement.
An operation is a postfix merge if the range of data ends at the bottom of the partition. You can specify only the TO NEXT PARTITION clause. The split partition cannot be the last partition (the rightmost partition in the list). on page 2-279	Compiles an SQL statement for later execution with EXECUTE.

Embedded-Only SQL/MX Statements

For more information on SQL/MX statements that you can use only in embedded SQL programs, see Section 3, Embedded-Only SQL/MX Statements.

Resource Control and Optimization Statements

Use these statements to control access to an SQL/MX table and its indexes and to catalogs on remote nodes:

LOCK TABLE Statement on page 2-268	Locks the specified table (or the underlying tables of a view) and its associated indexes for the duration of the active transaction.
REGISTER CATALOG Statement on page 2-315	Registers a catalog on a remote node.
UNLOCK TABLE Statement on page 2-383	Releases locks held on the specified nonaudited table or view.
UNREGISTER CATALOG Statement on page 2-384	Removes an empty catalog reference from a node.
UPDATE STATISTICS Statement on page 2-402	Updates statistics about the contents of a table and its indexes.

Control Statements

Use these statements to control the execution default options, plans, and performance of DML statements:

CONTROL QUERY DEFAULT Statement on page 2-60	Overrides the contents of the SYSTEM_DEFAULTS table for the current session.

CONTROL QUERY SHAPE Statement on page 2-62	Forces access plans by modifying the operator tree for a prepared statement.
CONTROL TABLE Statement on page 2-74	Specifies a performance-related option for DML accesses to a table or view.
SET TABLE TIMEOUT Statement on page 2-372	Specifies a dynamic timeout value in the run-time environment of the current session.

Object Naming Statements

Use these statements to set the value of the NAMETYPE attribute, which determines whether the object naming is ANSI or NSK for the current session, and to specify default ANSI names for the catalog and schema or Guardian physical names for the volume and subvolume:

SET CATALOG Statement on page 2-366	Sets the default ANSI catalog for unqualified schema names for the current session.
SET MPLOC Statement on page 2-368	Sets the default operating system volume and subvolume for SQL/MP physical object names for the current session.
SET NAMETYPE Statement on page 2-369	Sets the default NAMETYPE attribute value to ANSI or NSK for the current session.
SET SCHEMA Statement on page 2-370	Sets the default ANSI schema for unqualified object names for the current session.

Alias Statements

Use the following statements to manage mappings between logical and physical names for SQL/MP objects:

ALTER SEQUENCE Statement on page 2-13	Maps an existing alias to a different SQL/MP table.
CREATE SQLMP ALIAS Statement on page 2-104	Creates mappings from logical names to physical names for SQL/MP database objects.
DROP SQLMP ALIAS Statement on page 2-188	Drops mappings from logical names to physical names for SQL/MP database objects.

Stored Procedure Statements

Use these statements to register and invoke stored procedures in Java (SPJs):

CALL Statement on page 2-54	Initiates the execution of a stored procedure in Java (SPJ) in NonStop SQL/MX.
CREATE PROCEDURE Statement on page 2-88	Defines an existing Java method as an SPJ and registers it in NonStop SQL/MX.
DROP PROCEDURE Statement on page 2-182	Drops an SPJ and its stored procedure label from NonStop SQL/MX.
GRANT EXECUTE Statement on page 2-246	Grants access privileges for a procedure to specified users.
REVOKE EXECUTE Statement on page 2-323	Revokes access privileges for a procedure to specified users.

Note. The Result Set Support for Stored Procedures in Java is available only on systems running J06.05 and later J-series RVUs and H06.16 and later H-series RVUs. This feature is supported by SQL/MX 2.3.2 onwards.

Trigger Statements

Use these statements to create and manipulate triggers on SQL/MX tables:

ALTER TRIGGER Statement on page 2-48	Alters a trigger.
CREATE TRIGGER Statement on page 2-144	Creates a trigger.
DROP TRIGGER Statement on page 2-192	Drops a trigger.
SET Statement on page 2-365	Controls the action of a BEFORE trigger.
SIGNAL SQLSTATE Statement on page 2-381	Enables a trigger execution to raise an exception that causes both the triggered and triggering statements to fail.

Utilities

Use these utilities to transform metadata, duplicate tables, partition management, load indexes, purge and recover data from the database objects:

DOWNGRADE Utility on page 2-175	Transforms metadata from the existing version to a specified, lower version.
DUP Utility on page 2-194	Duplicates SQL/MX tables.
FASTCOPY Utility on page 2-226	Copies rows from one table or index into an existing compatible table or index.
MODIFY Utility on page 2-271	Performs partition management operations and Guardian rename operations on tables or indexes.
POPULATE INDEX Utility on page 2-304	Loads indexes.
PURGEDATA Utility on page 2-307	Purges data from tables, indexes, or partitions.
RECOVER Utility on page 2-311	Determines the state of a failed utility operation and restores recoverable objects. Also, recovers a schema from a failed change of ownership operation. For more information, see RECOVER SCHEMA Operation.
UPGRADE Utility on page 2-412	Transforms metadata from the existing version to the current schema version for the SQL/MX Software Version (MXV).

Privileges Required to Execute Utilities

Utility	Privileges Required
DOWNGRADE	Super ID.
DUP	SELECT privilege on the source table.
	Be the owner of the schema where the target table resides.
FASTCOPY	SELECT privilege on the source table.
	SELECT, INSERT, UPDATE and DELETE privilege on the target table.
MODIFY	Schema owner or object owner or super ID.
POPULATE INDEX	Have ALL privileges.
PURGEDATA	Have ALL privileges.
RECOVER	Have the corresponding utility privileges for the utility to be recovered.
RECOVER SCHEMA	Schema owner or the security administrator.
UPGRADE	Super ID.

Checking DDL Locks

Many utilities, for example, DUP, MODIFY, and POPULATE INDEX, lock both metadata and user data during the operation. The following are three kinds of locks:

- Transactional locks on metadata and user data rows

  A utility runs in multiple TMF transactions, managed by the utility itself. If the utility fails before completion, then TMF will back out the latest of those transactions and then release the lock.

- Non-transactional file locks on user data

  Some utilities must hold shared locks on user data outside TMF transactions, to prevent concurrent updates while the utility is reading the data. If the utility operation fails before completion, such locks are automatically released.

- Logical DDL Locks on metadata

  Most utilities require that the definition of the object being worked on remains stable for the duration of the utility operation. To ensure this, the utility inserts a so-called 'DDL Lock' into metadata, which prevents other DDL and utility operations from changing the definition of the object. If the utility operation fails before completion, the DDL Lock remains intact.

If a utility operation fails before completion, you must recover the following using the RECOVER utility:

- Work done in TMF transactions that are internally committed by the utility.

For example, a MODIFY TABLE utility may fail at a point where the target partition has been created but data movement has not yet started. Running RECOVER with the CANCEL option will then remove the target partition. Running RECOVER with the RESUME option performs the data movement to the created target partition.

- Removing the DDL Lock.

  Using the MODIFY TABLE example, MODIFY inserts a DDL Lock for the affected table. Running RECOVER removes the DDL Lock.

To find out whether a failed utility operation needs to be recovered, issue this query from an MXCI prompt:

```
select substring(o.object_name from 1 for 40)
    as object_name, o.object_type
   from
        nonstop_sqlmx_<system name>.system_schema.catsys c,
        nonstop_sqlmx_<system name>.system_schema.schemata s,
    <cat>.definition_schema_version_<version number>.objects o,
    <cat>.definition_schema_version_<version number>.ddl_locks d
     where c.cat_name = '<cat>'
       and c.cat_uid = s.cat_uid
       and s.schema_name = '<schema>'
       and s.schema_uid = o.schema_uid
       and o.object_name = '<table name>'
       and o.object_type = 'BT'
       and d.base_object_uid = o.object_uid;
```

You can use this query for indexes, replacing `<table name>` with `<index name>` and `'BT'` with `'IX'`.

You can use this query for sequence generators, replacing `<table name>` with `<sequence generator name>` and `'BT'` with `'SG'`.

A typical output of this query is:

```
OBJECT_NAME                                 OBJECT_TYPE
----------------------------------------    -----------

EMPLOYEES                                   BT

--- 1 row(s) selected.
>>
```

In this example, EMPLOYEES is the name of the object. You need to run the RECOVER utility.

If the query does not return rows, the failed partition operation has rolled back completely. You do not need to perform recovery.

ALTER INDEX Statement

Considerations for ALTER INDEX
Examples of ALTER INDEX

The ALTER INDEX statement modifies an SQL/MX index by changing one or more file attributes of the index or by renaming the index. See Database Object Names on page 6-13.

ALTER INDEX is an SQL/MX extension.

```
ALTER INDEX [[catalog-name.]schema-name.]index alter-action

alter-action is:
RENAME TO new-index-object-name
|ATTRIBUTE[S]attribute [,attribute ]...

attribute is:
     {ALLOCATE num-extents | DEALLOCATE}
     {AUDITCOMPRESS | NO AUDITCOMPRESS}
   | {CLEARONPURGE | NO CLEARONPURGE}
   | MAXEXTENTS num-extents
```

Syntax Description of ALTER INDEX

index

is the ANSI logical name of the index to alter, of the form:

[[*catalog-name*.]*schema-name*.]*index*

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56 and Database Object Names on page 6-13.

RENAME TO *new-index-object-name*

changes the logical name of the index within the schema.

new-index-object-name

specifies the new ANSI name of the index. The new ANSI name of the index cannot be qualified. However, the renamed index will remain in the current catalog and schema.

Note. RENAME TO changes the redefinition timestamp of the affected index and the associated base table. However, the remaining indexes of the table are not affected.

`ATTRIBUTE[S]` *`attribute`* `[,`*`attribute`* `]...`

changes the values of file attributes for the index:

ALLOCATE/DEALLOCATE on page 9-2	Controls amount of disk space allocated.
AUDITCOMPRESS on page 9-3	Controls whether unchanged columns occur in audit records.
CLEARONPURGE on page 9-5	Controls disk erasure when index is dropped.
MAXEXTENTS on page 9-7	Controls maximum disk space to be allocated.

In an ATTRIBUTES clause within a PARTITION clause, you must separate *`attributes`* with a space. In ATTRIBUTES clauses in other places, you can separate *`attributes`* with either a space or a comma.

For more detail, see the entry for a specific attribute.

Considerations for ALTER INDEX

You cannot use ALTER INDEX to change the Guardian name of a partition. To change the Guardian name of a partition, use the Modify utility with the rename option. For more information, see Renaming Guardian Location of Partitions of Tables, Indexes or Sequence Generators.

Authorization and Availability Requirements

To alter an index, you must own the schema or be the super ID or object owner.

All partitions of the index must be available when ALTER INDEX executes. The appropriate metadata tables must also be available.

Renaming an Index

You can use the rename option to change the name of an index. The following are the prerequisites for renaming an index:

- The new ANSI name must not already exist within the schema.
- All partitions of the index and base table must be available.

Effects on TMF

Rename

For information on effects of rename option on TMF, see Effects on TMF on page 2-40.

Effects on RDF

Rename

For information on effects of rename option on RDF, see Effects on RDF on page 2-40.

Examples of ALTER INDEX

- This example changes the maximum number of extents to 760:

  ```
  ALTER INDEX xempname ATTRIBUTE MAXEXTENTS 760
  ```

- The following command renames the index, `CAT.SCH.I1 to I2`:

  ```
  ALTER INDEX CAT.SCH.I1 RENAME TO I2;
  ```

  The new ANSI name of the index, `I2` is not fully qualified since the index continues to remain in the same catalog and schema as `I1`. After the index is renamed, the fully qualified name of the index is `CAT.SCH.I2`.

ALTER SEQUENCE Statement

Considerations for ALTER SEQUENCE

Examples of ALTER SEQUENCE

The ALTER SEQUENCE statement alters the ANSI name, INCREMENT BY value, MAXVALUE, MINVALUE and CYCLE attributes of a sequence generator.

Altering a sequence generator attribute changes the redefinition timestamp for the sequence generator. As a result, the runtime similarity checks fail for applications using altered sequence generators and the applications are recompiled. To avoid this runtime recompilation overhead, HP recommends recompiling all the applications using the sequence generator after altering the attributes.

```
ALTER SEQUENCE sequence alter-actions

alter-actions is:
{ sequence-generator-options
  | sequence-generator-restart-with
  | RENAME rename-clause
}

sequence-generator-options is:
{ sequence-generator-option [sequence-generator-option
…]
}

sequence-generator-option is:
{ sequence-generator-increment-by-option
  | sequence-generator-maxvalue-option
  | sequence-generator-minvalue-option
  | sequence-generator-cycle-option
}

sequence-generator-increment-by-option is:
  INCREMENT BY sequence-generator-numeric-value

sequence-generator-maxvalue-option is:
{ MAXVALUE sequence-generator-numeric-value
  | NOMAXVALUE
  | NO MAXVALUE
}

sequence-generator-minvalue-option is:
{ MINVALUE sequence-generator-numeric-value
  | NOMINVALUE
  | NO MINVALUE
}

sequence-generator-cycle-option is:
{ CYCLE
  | NOCYCLE
  | NO CYCLE
}

sequence-generator-restart-with is:
RESTART WITH sequence-generator-numeric-value

sequence-generator-numeric-value is:
< [+|-] numeric-literal >

rename-clause is:
TO < new-sequence-name >
```

Syntax Description of ALTER SEQUENCE

sequence

specifies the ANSI name of a sequence generator.

sequence-generator-increment-by-option

alters the INCREMENT BY value to the user specified value. This value must be less than the difference between the MAXVALUE and MINVALUE for the sequence generator, and cannot be zero.

sequence-generator-maxvalue-option

alters the MAXVALUE to the user specified value. This value must satisfy the following conditions:

- It cannot be less than the CURRENT_VALUE and MINVALUE
- It must be greater than the START WITH value

If NOMAXVALUE or NO MAXVALUE is specified, the MAXVALUE is set to the maximum value of the sequence generator data type.

sequence-generator-minvalue-option

alters the MINVALUE to the user specified value. This value must satisfy the following conditions:

- It must be less than or equal to the CURRENT_VALUE and less than the MAXVALUE
- It must be less than or equal to the START WITH value

If NOMINVALUE or NO MINVALUE is specified, the MINVALUE is set to the minimum value of the sequence generator data type.

sequence-generator-cycle-option

alters the CYCLE attriibute to the user specified value.

If user specified value is NO CYCLE, an exception is raised for the next value after an ascending sequence reaches the maximum value or descending sequence reaches the minimum value.

If user specified value is CYCLE, the sequence generator loops over to generate the minimum value after an ascending sequence reaches the maximum value. The sequence generator loops over to generate the maximum value after a descending sequence reaches the minimum value.

sequence-generator-restart-with

resets the CURRENT_VALUE with the user specified value.

`numeric-literal`

an exact numeric literal corresponding to the sequence generator data type. It cannot be greater than the maximum value or less than the minimum value of the sequence generator data type.

`rename-clause`

alters the logical name of a sequence generator.

`new-sequence-name`

specifies the new ANSI name for the sequence generator. The new ANSI name cannot be qualified. The renamed sequence generator remains in the current catalog and schema.

Considerations for ALTER SEQUENCE

You cannot use the ALTER SEQUENCE statement to change the partition name. For more information, see Managing a Sequence Generator on page 2-288.

Authorization Requirements

The schema owner, the Super ID or the object owner can execute the ALTER SEQUENCE statement.

Restrictions

- You cannot alter START WITH, CACHE and ORDER attributes for a sequence generator.
- You cannot specify the INCREMENT BY, MAXVALUE or NOMAXVALUE or NO MAXVALUE, MINVALUE or NOMINVALUE or NO MINVALUE attributes more than once in the statement.
- You cannot use the RESTART WITH and RENAME options with other attributes in the statement.

Examples of ALTER SEQUENCE

- This example changes the MAXVALUE of `MYSEQ` sequence generator:

```
ALTER SEQUENCE MYSEQ MAXVALUE 1000;
```

- This example changes the INCREMENT BY value of `MYSEQ` sequence generator:

```
ALTER SEQUENCE MYSEQ INCREMENT BY 10;
```

- This example changes the sequence generator option to CYCLE:

```
ALTER SEQUENCE MYSEQ CYCLE;
```

ALTER SQLMP ALIAS Statement

Considerations for ALTER SQLMP ALIAS
Examples of ALTER SQLMP ALIAS

The ALTER SQLMP ALIAS statement maps an existing alias to a different SQL/MP table.

ALTER SQLMP ALIAS is an SQL/MX extension.

```
ALTER SQLMP ALIAS catalog.schema.object
   [\node.]$volume.subvol.filename
```

Syntax Description of ALTER SQLMP ALIAS

catalog.schema.object

is the alias name of an SQL/MP table or view. *catalog* and *schema* denote ANSI-defined catalog and schema, and *object* is a simple name for the table or view. If any part of the name is an SQL/MX reserved word, you must delimit it by enclosing it in double quotes. For example: `mycat."sql".myview`.

[*node*.]$*volume.subvol.filename*

is the fully qualified Guardian physical name of an SQL/MP table or view.

In this four-part name, *node* is the name of a node of a NonStop server, $*volume* is the name of a disk volume, *subvol* is the name of a subvolume, and *filename* is the name of an SQL/MP table or view. if any of the four parts of the name is an SQL/MX reserved word, you must delimit it by enclosing it in double quotes. Such delimited parts are not case-sensitive. For example: `$myvol."join".mytab`.

If you do not specify *node*, the default is the Guardian node named in the =_DEFAULTS define.

Considerations for ALTER SQLMP ALIAS

Usage Restrictions

If the specified alias does not exist or the specified Guardian file does not exist, NonStop SQL/MX returns an error. If the ALTER SQLMP ALIAS statement specifies a physical file name that is the same as the current alias mapping, NonStop SQL/MX returns a warning.

Security of Alias

To alter an existing SQL/MP alias, you must own its schema or be the super ID.

Late Bind

If you compile an application that uses an SQL/MP alias and later you change the SQL/MP alias to map to a different SQL/MP table, the SQL/MP table definition is no longer compatible with the definition used at compile time. As a result, you must manually recompile applications that use the alias. If the late bind does not find the SQL/MP table underlying the SQL/MP alias or if the SQL/MP table was moved, NonStop SQL/MX returns an error.

For more information, see the *SQL/MX Programming Manual for C and COBOL*.

Examples of ALTER SQLMP ALIAS

- This example changes the physical name of an SQL/MP table:

```
ALTER SQLMP ALIAS SAMDBCAT.PERSNL.EMPLOYEE
   \MYSYS.$SAMDB.PERSNL.NEWEMP
```

ALTER TABLE Statement

Considerations for ALTER TABLE
Examples of ALTER TABLE

The ALTER TABLE statement modifies an SQL/MX table by:

- Adding a column to the table
- Adding or dropping a constraint on the table
- Changing one or more file attributes for the table
- Renaming the table

```
ALTER TABLE table alter-action

alter-action is:
     ADD [COLUMN] column-definition
   | ADD [CONSTRAINT constraint] table-constraint
   | DROP CONSTRAINT constraint [RESTRICT | CASCADE]
   | ATTRIBUTE[S] attribute [,attribute]...
   | RENAME TO new-table-object-name [CASCADE]
   | ATTRIBUTE[S] attribute [,attribute]...
   | ALTER COLUMN column_name recalibrate-option
   | ALTER COLUMN column_name SET basic-sequence-generator-
option

column-definition is:
   column-name data-type
   [DEFAULT default]
   [HEADING 'heading-string' | NO HEADING]
   [[CONSTRAINT constraint] column-constraint]...

data-type is:
     CHAR[ACTER] [(length)[CHARACTERS]]
         [CHARACTER SET char-set-name] [COLLATE DEFAULT]
         [UPSHIFT]
   | PIC[TURE] X [(length)][CHARACTER SET char-set-name]
```

```
length)
        [CHARACTER SET char-set-name] [COLLATE DEFAULT]
        [UPSHIFT]
   | VARCHAR (length) [CHARACTER SET char-set-name]
        [COLLATE DEFAULT] [UPSHIFT]
   | NUMERIC [(precision [,scale])] [SIGNED|UNSIGNED]
   | NCHAR [(length) [CHARACTER SET char-set-name]
               [COLLATE DEFAULT] [UPSHIFT]
   | NCHAR VARYING(length) [CHARACTER SET char-set-name]
               [COLLATE DEFAULT] [UPSHIFT]
   | SMALLINT [SIGNED|UNSIGNED]
precision [,scale])] [SIGNED|UNSIGNED]
   | PIC[TURE] [S]{ 9(integer) [V[9(scale)]] | V9(scale) }
               [DISPLAY [SIGN IS LEADING] | COMP]
   | FLOAT [(precision)]
| REAL
   | DOUBLE PRECISION
   | DATE
   | TIME [(time-precision)]
   | TIMESTAMP [(timestamp-precision)]
   | INTERVAL { start-field TO end-field | single-field }
 is:
     literal
   | NULL
   | CURRENT_DATE
   | CURRENT_TIME
   | CURRENT_TIMESTAMP
   | {CURRENT_USER | USER}

column-constraint is:
     UNIQUE
   | PRIMARY KEY [ASC[ENDING] | DESC[ENDING]]
   | CHECK (condition)
   | REFERENCES ref-spec

table-constraint is:
     UNIQUE (column-list)
   | PRIMARY KEY (key-column-list)
   | CHECK (condition)
   | FOREIGN KEY (column-list) REFERENCES ref-spec

column-list is:
   column-name [,column-name]...

key-column-list is:
   column-name [ASC[ENDING] | DESC[ENDING]]
      [,column-name [ASC[ENDING] | DESC[ENDING]]...
```

```
   ref-spec is:
   referenced-table [(column-list)]

[referential triggered action]

referential triggered action is:
     update rule [delete rule]
   | delete rule [update rule]

update rule is: ON UPDATE referential action

delete rule is: ON DELETE referential action
referential action is:
     RESTRICT
   | NO ACTION
   | CASCADE
   | SET NULL
   | SET DEFAULT

attribute is:
{ALLOCATE num-extents | DEALLOCATE}
   | {AUDITCOMPRESS | NO AUDITCOMPRESS}
   | {CLEARONPURGE | NO CLEARONPURGE}
   | MAXEXTENTS num-extents

recalibrate-option is:
   | RECALIBRATE
   | RECALIBRATE TO signed-numeric-literal
   | RECALIBRATE TO signed-numeric-literal NO SELECT

basic-sequence-generator-option is:
     INCREMENT BY signed-numeric-literal
   | MAXVALUE signed-numeric-literal
```

Syntax Description of ALTER TABLE

table

specifies the name of the table to alter. For more information, see Database Object Names on page 6-13.

ADD [COLUMN] *column-definition*

adds a column to *table*.

The clauses for the *column-definition* are specified as:

column-name

specifies the name for the new column in the table. *Column-name* is an SQL identifier. *column-name* must be unique among column names in the table. If the column name is an SQL/MX reserved word, you must delimit it by enclosing it in double quotes. For example: `mycat."sql".myview`. See Identifiers on page 6-56.

`data-type`

specifies the name and data type for the new column in the table.

`data-type` is the data type of the values that can be stored in *`column`*. See Data Types on page 6-17.

`DEFAULT` *`default`*

specifies a default value for the column. The added column must have a default value. You can declare the default value explicitly by using the DEFAULT clause or you can enable null to be used as the default by omitting both the DEFAULT and NOT NULL clauses. If you omit the DEFAULT clause and specify NOT NULL, NonStop SQL/MX returns an error. For existing rows of the table, the added column takes on its default value.

If you set the default to the datetime value CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP, NonStop SQL/MX uses January 1, 1 A.D. 12:00:00.000000 as the default date and time for the existing rows.

For any row that is added after the column is added, if no value is specified for the column as part of the add row operation, the column will receive a default value based on the current timestamp at the time the row is added.

If you set the default value to USER, CURRENT_USER, or SESSION_USER, NonStop SQL/MX uses " " (blank) as the default value for the existing rows.

For any row that is added after the column is added, if no value is specified for the column as part of the add row operation, the column will receive the current Guardian user ID for its value. See DEFAULT Clause on page 7-2.

`HEADING '`*`heading-string`*`'| NO HEADING`

specifies a string *`heading-string`* of 0 to 128 characters to use as a heading for the column if it is displayed with a SELECT statement in MXCI. The *`heading-string`* can contain characters only from the ISO88591 character set. The default heading is *`column`*, the column name. If you specify a heading that is identical to the column name, INVOKE and SHOWDDL do not display that heading.

If you specify NO HEADING or HEADING ‘’, NonStop SQL/MX stores this as HEADING ‘’, and the column name is displayed as the heading in a SELECT statement. The behavior for HEADING ‘’ is different from that of NonStop SQL/MP, which does not display anything for a heading in a SELECT statement if the heading is specified as HEADING ‘’.

[CONSTRAINT *constraint*] *column-constraint*

specifies a name *constraint* and constraint definition for a column constraint. See Database Object Names on page 6-13.

ADD [CONSTRAINT *constraint*] *table-constraint*

adds a constraint to the table and optionally specifies *constraint* as the name for the constraint. The new constraint must be consistent with any data already present in the table.

CONSTRAINT *constraint*

specifies a name for the column or table constraint. *constraint* must have the same catalog and schema as *table* and must be unique among constraint names in that schema. If you omit the catalog portion or the catalog and schema portions of the name you specify in *constraint*, NonStop SQL/MX expands the name by using the catalog and schema for *table*. See Database Object Names on page 6-13.

If you do not specify a constraint name, NonStop SQL/MX constructs an SQL identifier as the name for the constraint in the catalog and schema for *table*. The identifier consists of the fully qualified table name concatenated with a system-generated unique identifier. For example, a constraint on table A.B.C might be assigned a name such as A.B.C_971..._01... . .

UNIQUE
or
UNIQUE *column-list*

is a column or table constraint (respectively) that specifies that the column or set of columns cannot contain more than one occurrence of the same value or set of values. If you omit UNIQUE, duplicate values are allowed.

column-list cannot include more than one occurrence of the same column. In addition, the set of columns you specify on a UNIQUE constraint cannot match the set of columns on any other UNIQUE constraint for the table or on the PRIMARY KEY constraint for the table. Columns you define as unique must be specified as NOT NULL.

A UNIQUE constraint is enforced with a unique index. If there is already a user-defined unique index on *column-list*, NonStop SQL/MX uses this index; if not, the system creates a unique index.

The maximum combined length of the columns for a UNIQUE constraint is 2010 bytes for 4K blocks and 2048 bytes for 32K blocks.

`PRIMARY KEY [ASC[ENDING] | DESC[ENDING]] [[NOT] DROPPABLE]`
or
`PRIMARY KEY` *`key-column-list`*

is a column or table constraint (respectively) that specifies a column or set of columns as the primary key for the table. *`key-column-list`* cannot include more than one occurrence of the same column. In addition, the set of columns you specify on a PRIMARY KEY constraint cannot match the set of columns on any UNIQUE constraint for the table.

ASCENDING and DESCENDING specify the direction for entries in each column within the key. The default is ASCENDING.

The PRIMARY KEY value in each row of the table must be unique within the table. Columns within a PRIMARY KEY cannot contain nulls. A PRIMARY KEY defined for a set of columns implies that the column values are unique and not null.

When a PRIMARY KEY table constraint is added by using the ALTER TABLE statement, it is always droppable. For a PRIMARY KEY column constraint, you cannot specify NOT DROPPABLE; if you do, NonStop SQL/MX returns an error.

A PRIMARY KEY constraint is enforced with a unique index. If there is already a unique index on *`key-column-list`*, NonStop SQL/MX uses this index; if not, the system creates a unique index. Because the PRIMARY KEY constraint uses a supporting unique index, the clustering key is not part of the constraint definition and therefore the maximum combined length of the columns for the PRIMARY KEY is 2010 bytes for 4K blocks and 2048 bytes for 32K blocks.

When a PRIMARY KEY constraint is created on a table, all the index columns must have a NOT NULL clause in the CREATE TABLE statement for the table.

The value of the PRIMARY_KEY_CONSTRAINT_DROPPABLE_OPTION attribute in the DEFAULTS Table has no affect on a PRIMARY KEY constraint added by using the ALTER TABLE statement because in this case the PRIMARY KEY is always droppable.

`CHECK (`*`search-condition`*`)`

is a constraint that specifies a *`condition`* that must be satisfied for each row in the table.

NonStop SQL/MX checks the *`condition`* whenever an operation occurs that might affect its value. The operation is allowed if the predicate in the search condition evaluates to TRUE or null, but is prohibited if the predicate evaluates to FALSE. When a check constraint is added, existing data is checked for violations.

You cannot refer to the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP function in a CHECK constraint, and you cannot use subqueries in a CHECK constraint.

REFERENCES *ref-spec*

specifies a references column constraint. The maximum combined length of the columns for a REFERENCES constraint is 2010 bytes for 4K blocks and 2048 bytes for 32K blocks.

FOREIGN KEY (*column-list*) REFERENCES *ref-spec*

is a referential table constraint. A referential constraint for the table declares that a column or set of columns (called a foreign key) in *table* can contain only values that match those in a column or set of columns specified in the REFERENCES clause.

The two columns or sets of columns must have the same characteristics (data type, length, scale, precision), and there must be a UNIQUE or PRIMARY KEY constraint on the column or set of columns specified in the REFERENCES clause.

The foreign key is the column or set of columns specified in the FOREIGN KEY clause, immediately following the FOREIGN KEY keywords.

A FOREIGN KEY constraint is enforced with a nonunique index. If there is already a unique or nonunique index on *key-column-list*, NonStop SQL/MX uses this index; if not, it creates a nonunique index.

ref-spec is:

referenced-table [(*column-list*)] [*referential triggered action*]

referenced-table is the table referenced by the foreign key in a referential constraint. *referenced-table* cannot be a view, and *referenced-table* cannot be the same as *table*.

column-list specifies the column or set of columns in *referenced-table* that corresponds to the foreign key in *table*. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column list associated with FOREIGN KEY. If *column-list* is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

update rule specifies what *referential action* is taken when *column-list* in *referenced-table* is updated. If no ON UPDATE clause is specified, a default of ON UPDATE NO ACTION is assumed.

delete rule specifies what *referential action* is taken when a row in *referenced-table* is deleted. If no ON DELETE clause is specified, a default of ON DELETE NO ACTION is assumed.

referential action

RESTRICT *referential action* means that the referential check is made for each row. An error is raised when the referential constraint is violated.

ANSI SQL-99 standard: NO ACTION *referential action* means that the referential check is made at the end of the SQL statement. An error is raised when the referential constraint is violated.

NonStop SQL/MX does not support NO ACTION referential action in the way it is specified by ANSI SQL-99. However, you can change NO ACTION's behavior to be the same as RESTRICT by setting an appropriate value for the Control Query Default REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT. Options for this attribute are:

OFF	SQL issues an error.
SYSTEM	SQL issues warning 1302, indicating that it will behave like RESTRICT. The default is SYSTEM.
ON	Makes NO ACTION behave like RESTRICT, without warnings or errors.

When CASCADE is specified with the ON DELETE referential triggered action, a row in the referencing table and its corresponding row in the *referenced-table* is deleted. This maintains consistency between the referencing and referenced tables.

When SET NULL is specified with the ON DELETE referential triggered action, and a row from the referencing table matches the row in the *referenced-table*, the referencing column(s) of the referencing row from the referencing table is set to NULL.

When SET DEFAULT is specified with the ON DELETE referential triggered action, and a row from the referencing table matches the row in the *referenced-table*, the referencing column(s) of the referencing row from the referencing table is set to its DEFAULT value.

When CASCADE is specified with the ON UPDATE referential triggered action, a row in the referencing table and its corresponding row in the *referenced-table* is updated.

When SET NULL is specified with the ON UPDATE referential triggered action, and a row in the referencing table matches the row in the *referenced-table*, the referencing column(s) of the referencing row from the referencing table is set to NULL.

When SET DEFAULT is specified with the ON UPDATE referential triggered action, and a row in the referencing table matches the row in the *referenced-table*, the referencing column(s) of the referencing row from the referencing table is set to its DEFAULT value.

Note. The referential actions CASCADE, SET NULL, and SET DEFAULT are available only on systems running J06.09 and later J-series RVUs and H06.20 and later H-series RVUs.

A table can have an unlimited number of referential constraints, and you can specify the same foreign key in more than one referential constraint, but you must define each referential constraint separately.

`DROP CONSTRAINT` *`constraint`* `[RESTRICT | CASCADE]`

drops a constraint from the table. The constraint name *constraint* must be specified. If you did not specify a name when you created the constraint or do not know the constraint name, you can use SHOWDDL to display it.

A referential constraint is dependent on its referenced column list. This column list is associated with a UNIQUE or PRIMARY KEY constraint. When a UNIQUE or PRIMARY KEY constraint is dropped, NonStop SQL/MX checks if any referential constraints are dependent on the constraint.

If you specify RESTRICT and referential constraints are dependent on the constraint, you cannot drop the constraint.

If you specify CASCADE and referential constraints are dependent on the constraint, those dependent constraints are dropped in addition to the specified constraint being dropped.

If you drop a constraint, NonStop SQL/MX drops its dependent index if SQL/MX originally created the same index. If the constraint uses an existing index, the index is not dropped.

The default is RESTRICT.

`CONSTRAINT` *`constraint`*

specifies a name for the column or table constraint. *constraint* must have the same catalog and schema as *table* and must be unique among constraint names in that schema. If you omit the catalog portion or the catalog and schema portions of the name you specify in *constraint*, NonStop SQL/MX expands the name by using the catalog and schema for *table*. See Database Object Names on page 6-13.

If you do not specify a constraint name, NonStop SQL/MX constructs an SQL identifier as the name for the constraint in the catalog and schema for *table.* The identifier consists of the fully qualified table name concatenated with a system-generated unique identifier. For example, a constraint on table A.B.C might be assigned a name such as A.B.C_971..._01... .

`UNIQUE`
or
`UNIQUE (`*`column-list`*`)`

is a column or table constraint (respectively) that specifies that the column or set of columns cannot contain more than one occurrence of the same value or set of values. If you omit UNIQUE, duplicate values are allowed unless the column is part of the PRIMARY KEY.

`column-list` cannot include more than one occurrence of the same column. In addition, the set of columns you specify on a UNIQUE constraint cannot match the set of columns on any other UNIQUE constraint for the table or on the PRIMARY KEY constraint for the table. All columns defined as unique must be specified as NOT NULL.

A UNIQUE constraint is enforced with a unique index. If there is already a unique index on *`column-list`*, NonStop SQL/MX uses this index; if not, the system creates a unique index.

`PRIMARY KEY [ASC[ENDING] | DESC[ENDING]] DROPPABLE]`
or
`PRIMARY KEY `*`key-column-list`*

is a column or table constraint (respectively) that specifies a column or set of columns as the primary key for the table. *`key-column-list`* cannot include more than one occurrence of the same column. In addition, the set of columns you specify on a PRIMARY KEY constraint cannot match the set of columns on any UNIQUE constraint for the table.

ASCENDING and DESCENDING specify the direction for entries in each column within the key. The default is ASCENDING.

The PRIMARY KEY value in each row of the table must be unique within the table. Columns within a PRIMARY KEY cannot contain nulls. A PRIMARY KEY defined for a set of columns implies that the column values are unique and not null.

When a PRIMARY KEY table constraint is added by using the ALTER TABLE statement, it is always droppable. For a PRIMARY KEY column constraint, you cannot specify NOT DROPPABLE; if you do, NonStop SQL/MX returns an error.

A PRIMARY KEY constraint is enforced with a unique index. If there is already a user-defined unique index on *`key-column-list`*, NonStop SQL/MX uses this index; if not, it creates a unique index. Because the PRIMARY KEY constraint uses a supporting unique index, the clustering key is not part of the constraint definition. Therefore, the maximum combined length of the columns for the PRIMARY KEY is 2010 bytes for 4K blocks and 2048 bytes for 32K blocks.

When a PRIMARY KEY constraint is created on a table, all the index columns must have a NOT NULL clause in the CREATE TABLE statement for the table.

The value of the PRIMARY_KEY_CONSTRAINT_DROPPABLE_OPTION attribute in the DEFAULTS Table has no affect on a PRIMARY KEY constraint added by using the ALTER TABLE statement because in this case the PRIMARY KEY is always droppable.

CHECK (*condition*)

is a constraint that specifies a *condition* that must be satisfied for each row in the table.

NonStop SQL/MX checks the *condition* whenever an insert or update operation occurs that might affect its value. The operation is allowed if the predicate in the search condition evaluates to TRUE or null, but is prohibited if the predicate evaluates to FALSE. When a check constraint is added, existing data is checked for violations.

You cannot refer to the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP function in a CHECK constraint, and you cannot use subqueries in a CHECK constraint.

See Search Condition on page 6-108.

REFERENCES *ref-spec*
  or
FOREIGN KEY (*column-list*) REFERENCES *ref-spec*

is a column or table constraint (respectively) that specifies a referential constraint for the table, declaring that a column or set of columns (called a foreign key) in *table* can contain only values that match those in a column or set of columns specified in the REFERENCES clause.

The two columns or sets of columns must have the same characteristics (data type, length, scale, precision), and there must be a UNIQUE or PRIMARY KEY constraint on the column or set of columns specified in the REFERENCES clause.

The foreign key is the column or set of columns specified in the FOREIGN KEY clause, immediately following the FOREIGN KEY keywords.

ref-spec is:

referenced-table [(*column-list*)]

referenced-table is the table referenced by the foreign key in a referential constraint. *referenced-table* cannot be a view, and *referenced-table* cannot be the same as *table*.

column-list specifies the column or set of columns in *referenced-table* that corresponds to the foreign key in *table*. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column list associated with FOREIGN KEY. If *column-list* is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

A table can have an unlimited number of referential constraints and you can specify the same foreign key in more than one referential constraint, but you must define each referential constraint separately.

Publish/Subscribe's embedded update and embedded delete statements are not allowed on tables with referential integrity constraints.

You cannot create self-referencing foreign key constraints. When a foreign key constraint is added to an existing table, NonStop SQL/MX verifies that the data does not violate the constraint. If it does, a message is returned indicating the constraint was not created.

`RENAME TO` *`new-table-object-name`* `[CASCADE]`

changes the logical name of the table within the schema.

`new-table-object-name`

specifies the new ANSI name of the table. The new ANSI name of the table cannot be qualified. However, the renamed table will remain in the current catalog and schema.

`CASCADE`

specifies that the system generated ANSI names of indexes and constraints associated with the table will be renamed automatically.

Note. The CASCADE option renames the system-generated constraints and indexes associated with the table. System-generated names have the following form:
<ANSI name prefix>*_nnnnnnnnn_nnnn*
where, *n* is a digit.
CASCADE option matches the ANSI name prefix of constraints with the table name. The CASCADE option does not rename the constraints whose ANSI name prefix does not match the table name. However, CASCADE option renames the system generated index, irrespective of the name. Intermittent use of CASCADE option in rename operations might lead to unintended results. Therefore, HP recommends that you either use CASCADE continuously or do not use this option.

Note.

- Renaming a table changes the text of referencing views, RI constraints, and triggers to reference the new name.
- RENAME TO changes the redefinition timestamp of the following objects:
  - The affected table.
  - All indexes on the table.
  - All the tables referenced by the affected table.
  - All the tables referencing the affected table.

`ATTRIBUTE[S] attribute [,attribute ]...`

changes the values of file attributes for the table and its dependent indexes. You can separate *attributes* with either a space or a comma. File attributes you can specify are:

ALLOCATE/DEALLOCATE on page 9-2	Controls amount of disk space allocated.
AUDITCOMPRESS on page 9-3	Controls whether unchanged columns are included in audit records.
CLEARONPURGE on page 9-5	Controls disk erasure when table is dropped.
MAXEXTENTS on page 9-7	Controls maximum disk space to be allocated.

Unlike NonStop SQL/MP's form of this statement, SQL/MX's ALTER TABLE statement has no PARTONLY clause. When you supply a new value for attributes, ALTER TABLE modifies the value of the attribute on all partitions of the table. For more detail, see the entry for a specific attribute.

basic-sequence-generator-option

`INCREMENT BY signed-numeric-literal`

See INCREMENT BY *signed-numeric-literal* in Syntax Description of CREATE TABLE on page 2-111.

`MAXVALUE signed-numeric-literal`

See MAXVALUE *signed-numeric-literal* in Syntax Description of CREATE TABLE on page 2-111. NO MAXVALUE is not a valid specification for ALTER TABLE ALTER COLUMN.

recalibrate-option

recalibrates the `CURRENT_VALUE` of the internal Sequence Generator. This might be necessary after a FASTCOPY, DUP or an IMPORT operation. There are three ways to recalibrate:

- Recalibrate based on the INCREMENT BY internal sequence generator option and the maximum value of the IDENTITY column in the base table.
- Recalibrate to a user-specified value with a SELECT to obtain the maximum value of the IDENTITY column from the base table.
- Recalibrate to a user-specified value without performing a SELECT of the IDENTITY column maximum value.

See Recalibrating the Sequence Generator of an IDENTITY column on page 2-38 for more details.

Considerations for ALTER TABLE

You cannot use ALTER TABLE to change a partition's name.

Effect of Adding a Column on View Definitions

The addition of a column to a table has no effect on existing view definitions. Implicit column references specified by SELECT * in view definitions are replaced by explicit column references when the definition clauses are originally evaluated.

Authorization and Availability Requirements

To alter a table, you must own the schema or be the super ID or object owner. You must also have access to all partitions of the table itself.

ALTER TABLE works only on user-created tables. You cannot use it to modify a metadata table even if you are the owner of the metadata tables or a SUPER user.

Adding a Constraint

To add a constraint that refers to a column in another table, you must have REFERENCES privileges on that column.

Dropping a Constraint

To drop a constraint, you must be the schema or object owner of the table on which the constraint has been defined or be the super ID. If you are owner of the table which the referential constraint is referencing you can revoke the REFERENCE privilege on the column. Revoking the REFERENCE privileges, in effect, drops the constraint. You can revoke the REFERENCE privilege with a REVOKE command or indirectly through a DROP TABLE … CASCADE statement.

Adding a Column

A user who has UPDATE or REFERENCES privileges on a table also has those privileges on added columns of the table.

Renaming a Table

You can use the rename option to change the name of a table. The following are the prerequisites for renaming a table:

- The new ANSI name must not already exist within the schema.
- The table and all of the partitions must be available. Also, all partitions of indexes of the table must be available.

Constraints Implemented With Indexes

NonStop SQL/MX uses unique indexes to implement all UNIQUE constraints, including PRIMARY KEY constraints. Nonunique indexes are used to implement the foreign key portion of all referential constraints added with ALTER TABLE.

When you add such a constraint, NonStop SQL/MX checks if an existing index can be used to implement the constraint and if not, automatically creates a new index (if possible, with the same name as the constraint). It uses the same primary extent size, secondary extent size and MAXEXTENTS values as the base table's primary partition. The index is created on the same volume as the base table's primary partition. NonStop SQL/MX then populates the new index.

After NonStop SQL/MX populates the index, you should perform a FUP RELOAD on the index and all its partitions, to organize the index structure more efficiently and to reduce index levels.

If you are creating a constraint on a large table, you might receive an error 45 (file full). In addition, because NonStop SQL/MX executes the creation of the constraint in a single TMF transaction, you might experience TMF limitations such as a full audit trail file or transaction timeout.

If you create an index with the default values by mistake, you might need to re-create the index. You can alter maxextents size after the index is created, but you cannot alter primary and secondary extent sizes. You can use MODIFY to partition the index and move the partitions to desired locations.

Indexes used to enforce constraints can require significant amounts of disk space, and NonStop SQL/MX might be unable to create the supporting index when you add the constraint. Consequently, the add constraint operation fails.

Note. When using a large table, you should create the supporting index before you create a constraint. As a result, you can create the index as needed (for example, with partitions) so that you have better control over use of disk volumes. To create the constraint, you must create a unique or nonunique index before retrying the ALTER TABLE ADD CONSTRAINT operation. You might also want to partition the supporting index for better performance.

Adding CHECK and FOREIGN KEY Constraints

When a CHECK or FOREIGN KEY constraint is added to a table containing data, the existing data is validated to ensure it conforms to the constraint. While this validation takes place, the table is locked for read-only access. For a FOREIGN KEY constraint, both the referencing and referenced tables are locked. This means that while SQL/MX can perform statement compilations that use these tables or perform read operation to these tables, you will not be able to update the data during validation.

A full-file scan that is run in a single TMF transaction could experience TMF limitations, such as transaction timeout, if a large amount of data is to be checked.

Dropping FOREIGN KEY Constraints

To drop a table’s foreign key, you must perform SHOWDDL on the table to find the constraint’s system identification, then use that value in the ALTER TABLE statement. For a description of SHOWDDL, see SHOWDDL Command on page 4-83. For an example of an ALTER TABLE statement to drop a foreign key, see Examples of ALTER TABLE.

Altering the MAXVALUE and INCREMENT BY options on IDENTITY columns

The only options that can be altered for an IDENTITY column are:

- INCREMENT BY
- MAXVALUE

These rules apply:

- Only one IDENTITY column sequence generator option can be altered at a time.
- The INCREMENT BY option cannot be 0 (zero), less than 0 (zero), and cannot be greater than the maximum value of the data type of the IDENTITY column.
- The INCREMENT BY or MAXVALUE options can be used only on an IDENTITY column.
- When the INCREMENT BY option is altered, only the attributes of the internal Sequence Generator are altered. The `CURRENT_VALUE` of the internal Sequence Generator is not altered. The current value is incremented using the INCREMENT BY value applied during previous INSERT operations. The next INSERT after the ALTER TABLE ALTER COLUMN command obtains the `CURRENT_VALUE`. The new INCREMENT BY value set by the ALTER TABLE ALTER COLUMN SET command will then be applied, creating a new current value.
- The MAXVALUE option cannot be 0 (zero), less than 0 (zero) or greater than the maximum value of the data type of the IDENTITY column.
- The MAXVALUE option must be greater than the `CURRENT_VALUE` of the internal Sequence Generator.
- The MAXVALUE option value cannot be less than the INCREMENT BY option value.
- A valid numeric value must be specified for the MAXVALUE option. NO MAXVALUE is not allowed. If you use NO MAXVALUE on the ALTER TABLE ALTER COLUMN specification, an error will be raised:

```
>>alter table T115T002 alter column id_key set NO MAXVALUE;

*** ERROR[1595] The MAXVALUE option for the IDENTITY column ID_KEY
must be a valid numeric value. NO MAXVALUE is not allowed.

--- SQL operation failed with errors.
```

- Similarity check for an INSERT query that involves a system-generated IDENTITY column value will fail; the query must be recompiled.

Altering the MAXVALUE attribute on IDENTITY columns

When the MAXVALUE maximum is reached for the IDENTITY column, error ERROR[8934] is raised that says the maximum has been exceeded. Subsequent INSERTs fail with ERROR[8934], but the `CURRENT_VALUE` of the internal Sequence

Generator is updated for every failed INSERT. If an artificially low cycle range is created by having the MAXVALUE set to a value lower than the natural maximum allowed for the data type, the ALTER TABLE ALTER COLUMN SET MAXVALUE option can then be used to raise the maximum, up to the natural maximum allowed for the data type. This allows for more available values in the cycle range for the internal Sequence Generator. Inserts are successful until the new MAXVALUE is reached.

IDENTITY column and redefinition timestamp

When the MAXVALUE or the INCREMENT BY attribute is altered, the redefinition timestamp is updated for the base table that contains the IDENTITY column as well as for the SG Table associated with that IDENTITY column. The following example illustrates this behavior:

```
control query default SEQUENCE_GENERATOR_CACHE '1';
--- SQL operation complete.

control query default AUTOMATIC_RECOMPILATION 'ON';
--- SQL operation complete.

control query default RECOMPILATION_WARNINGS 'ON';
--- SQL operation complete.

create table T115T009 (id_key int unsigned GENERATED BY DEFAULT AS
IDENTITY (MINVALUE 1 MAXVALUE 2)
   , b int
   , c int);
--- SQL operation complete.

prepare s1 from insert into T115T009 values (DEFAULT, 1, 1);
--- SQL command prepared.

execute s1;
--- 1 row(s) inserted.

execute s1;
--- 1 row(s) inserted.

-- Two successful executes and the third execute Must get
--"*** ERROR[8934] The MAXVALUE for the sequence generator has been
```

```
exceeded."
-- Value 3 exceeded MAXVALUE 2.

execute s1;

*** ERROR[8934] The MAXVALUE for the sequence generator has been
exceeded.

--- 0 row(s) inserted.

-- Note: The value from the third execute is lost since the execute
failed.

select * from T115T009;

id_KEY              B            C

-------------  -----------  -----------


    1               1            1

    2               1            1


--- 2 row(s) selected.

-- Recover from the 8934 error by altering the MAXVALUE attribute of
the sequence generator.

alter table T115T009 alter column id_key set maxvalue 100;
--- SQL operation complete.

-- INSERT queries involving IDENTITY column are
-- non-retryable queries similar to INSERT queries

-- involving tables with indexes. So, open to the SG Table -- is blown
away and the query is not auto-recompiled.
```

```
--- Retry happens on the subsequent execute that raises
---   *** ERROR[8574] An OPEN was blown away on SG_table.

execute s1;

*** ERROR[8574] An OPEN was blown away on table
CAT.S15."@@INTERNAL_SG_511312700289154_@@".

*** ERROR[8935] The sequence generator update failed, see additional
diagnostics for failure details.

--- 0 row(s) inserted.

-- The above error occurs on the access to the SG Table,
-- hence the CURRENT_VALUE in the SG Table is not updated.

select * from T115T009;

ID_KEY          B           C

----------  ----------  -----------


   1            1            1

   2            1            1


--- 2 row(s) selected.

-- This is a non-retriable query, hence execute command –
-- must be re-issued.
-- The query is recompiled and the following warning is
-- raised "*** WARNING[8576] Statement was recompiled"
--
execute s1;

*** WARNING[8576] Statement was recompiled.

--- 1 row(s) inserted.

-- Notice the value 3 is missing from the ID_KEY value.
-- It was lost during the
-- execute that failed with " *** ERROR[8934] The MAXVALUE
-- for the sequence generator has been exceeded."

select * from T115T009;

ID_KEY            B            C

-------------  -----------  -----------


    1             1            1

    2             1            1

    4             1            1

--- 3 row(s) selected.
```

Recalibrating the Sequence Generator of an IDENTITY column

Recalibrate an IDENTITY column based on the INCREMENT BY value

This method always performs a SELECT on the base table containing the IDENTITY column to obtain the current maximum value of the IDENTITY column. This maximum value is incremented using the value of the INCREMENT BY internal Sequence Generator option. It is checked with the current numbering scheme of the INCREMENT BY option. If the newly incremented maximum value is not consistent with the numbering scheme, the value will be increased to the next value that would be consistent with the INCREMENT BY numbering scheme.

Rules for recalibrating based on the INCREMENT BY value

- The column to be recalibrated must exist and be an IDENTITY column in the table.
- If the table containing the IDENTITY column was newly created and has no rows added, no calibration is necessary. The recalibration statement ends successfully, but no update will be performed for the internal Sequence Generator current value.
- If the table containing the IDENTITY column has rows added, calibration might be necessary. The internal Sequence Generator `CURRENT_VALUE` column is recalibrated to a number larger than the maximum value of the IDENTITY column in the base table.
  - The current maximum value for the IDENTITY column is obtained by performing a SELECT on the IDENTITY column. This maximum value is added to the INCREMENT BY value of the internal Sequence Generator option. This incremented value is saved in the `CURRENT_VALUE` of the internal Sequence Generator table (SG Table).
  - When the new `CURRENT_VALUE` is calculated for a `RECALIBRATE` command, the new `CURRENT_VALUE` must not be greater than the maximum value allowed for the data type of the IDENTITY column, or the MAXVALUE internal Sequence Generator option value. An error is returned if the calculated new `CURRENT_VALUE` exceeds these maximums and the `CURRENT_VALUE` of the internal Sequence Generator table remains unmodified.
  - The INCREMENT BY numbering scheme will be honored when determining the new value for the `CURRENT_VALUE` of the internal Sequence Generator. When the new current value is calculated, it will be compared to determine if it matches the next logical number that would be consistent with the numbering scheme for the INCREMENT BY value. If the number is not consistent, then a number will be added to the current value to make it consistent. This synchronizes the new `CURRENT_VALUE` to the INCREMENT BY numbering scheme. For more information, see Example of ALTER TABLE ALTER COLUMN..RECALIBRATE on page 2-45.

- This incremented value is saved in the `CURRENT_VALUE` of the internal Sequence Generator table. This insures the internal Sequence Generator value will then generate unique numbers for the IDENTITY column.

Recalibrate to a user-specified value with SELECT

This method always performs a SELECT on the base table containing the IDENTITY column to obtain the current maximum value of the IDENTITY column. This maximum value will be compared to the user-specified value. The user-specified value will not be incremented using the INCREMENT BY internal Sequence Generator option or adjusted to match its numbering scheme.

Rules for recalibrating to a user-specified value

- The column to be recalibrated must exist and be an IDENTITY column in the table.
- The user-specified recalibration value must be included, be a positive number, and must not be greater than the maximum value allowed for the data type of the IDENTITY column. In addition, the user-specified recalibration value must not be greater than MAXVALUE option of the internal Sequence Generator.
- The current maximum value obtained by performing the SELECT is used in the following rules:
    - If the default specification type is GENERATED ALWAYS AS IDENTITY, the user-specified recalibration value must be larger than the current maximum value of the IDENTITY column in the base table.
    - If the default specification type is GENERATED BY DEFAULT AS IDENTITY, a number less than the current maximum value of the IDENTITY column is allowed.
- The user-specified recalibration value must not be less than the START WITH and MINVALUE options of the internal Sequence Generator.
- The user-specified recalibration value will not be synchronized with the INCREMENT BY numbering scheme. The user-specified value will be considered a pure override.

Recalibrate to a user-specified value without SELECT

This method does not perform a SELECT on the base table containing the IDENTITY column to obtain the current maximum value of the IDENTITY column. The user-specified value will not be incremented using the INCREMENT BY internal Sequence Generator option or adjusted to match its numbering scheme.

Rules for recalibrating to a user-specified value without SELECT

- The column to be recalibrated must exist and be an IDENTITY column in the table.
- The user-specified recalibration value must be included, be a positive number, and must not be greater than the maximum value allowed for the data type of the IDENTITY column.

- The user-specified recalibration value must not be greater than MAXVALUE option of the internal Sequence Generator.
- The default specification type must be GENERATED BY DEFAULT AS IDENTITY. ERROR[1601] is raised for default specification type GENERATED ALWAYS AS IDENTITY.
- The user-specified recalibration value must not be less than the START WITH and MINVALUE options of the internal Sequence Generator.
- The user-specified recalibration value will not be synchronized with the INCREMENT BY numbering scheme. The specified value will be considered a pure override.

SQL/MX Extensions to ALTER TABLE

The SQL/MX extensions are:

- ATTRIBUTES clause
- ASCENDING and DESCENDING options on the PRIMARY KEY constraint

Considerations for Referential Integrity

For information on referential integrity constraints, see the Considerations for Referential Integrity section in CREATE TABLE.

Effects on TMF

Rename

After ANSI rename operations are performed, TMF recovery including backout, volume, and file recovery are fully supported. TMF online dumps remain valid after ANSI rename. Use the new ANSI names to access these dumps, because you can no longer use the original ANSI names after rename is performed.

Effects on RDF

Rename

You must perform corresponding rename operations on the backup system to synchronize primary and backup systems. This ensures that the logical correspondence between the ANSI names are maintained in the primary and backup databases.

Effects on open blown away

Rename

ANSI rename changes the redefinition timestamp of a table. Therefore, if there is an open blown away after an ANSI rename operation, the similarity check will pass, because it is the same physical table with a different name. The following example illustrates this occurrence:

```
>>control query default RECOMPILATION_WARNINGS 'ON';
--- SQL operation complete.
>>prepare q from select * from t1;
--- SQL command prepared.
>>execute q;
<successful result>
>>alter table t1 rename to t2;
--- SQL operation complete.
>>execute q;
*** WARNING[8578] Similarity check passed.
<successful result>
>>
```

In this example, the first command `execute q,` uses the original ANSI name to access the table t1. The second command changes the name and redefinition timestamp of the table. The third command `execute q,` also uses the original ANSI name to access the table t1. Although the rename command changes the name and redefinition timestamp of the table, the similarity check passes because the table t1 was already open using the first command.

New attempts to access a renamed table using the original ANSI name fail, because the original ANSI name no longer exists. The following example illustrates this occurrence:

```
>>prepare q from select * from t1;
--- SQL command prepared.
>>alter table t1 rename to t2;
--- SQL operation complete.
>>execute q;
*** ERROR[1004] Object CAT.SCH.T1 does not exist or object type is
invalid for the current operation.
*** ERROR[8300] Late name resolution failed for table, view or
stored procedure CAT.SCH.T1.
--- 0 row(s) selected.
>>
```

In this example, the first command rename, changes the name and redefinition timestamp of the table t1. The second command `execute q`, tries to access the table t1 by the original ANSI name. An error is returned because the table t1 was renamed. There are two ways to avoid such a failure:

- Create a new table with the same ANSI name and layout. When an application accesses the table, the executor determines that the redefinition timestamp is changed but the ANSI name and layout of the table remains the same. Therefore, the similarity check passes.
- In NonStop SQL/MX, when an application uses the “prototyping” mechanism, it can specify the new ANSI name at runtime instead of the original ANSI name. The redefinition timestamp of the table changes but it remains as the same physical table with a different ANSI name. Therefore, the similarity check passes.

Examples of ALTER TABLE

- This example adds a UNIQUE table constraint:

```
ALTER TABLE persnl.project
  ADD CONSTRAINT projtimestamp_uc
    UNIQUE (projcode, ship_timestamp);
```

- This example drops a constraint:

```
ALTER TABLE persnl.project DROP CONSTRAINT projtimestamp_uc;
```

- This example adds a column with a foreign key constraint:

```
ALTER TABLE persnl.project
  ADD COLUMN projlead
    NUMERIC (4) UNSIGNED
    HEADING 'Project/Lead'
    CONSTRAINT projlead_fk REFERENCES persnl.employee;
```

- This example adds a foreign key table constraint. Note that if the foreign key is one column, you can include the constraint with the column definition, as in the preceding example.

```
ALTER TABLE persnl.project
  ADD CONSTRAINT projlead_fk
    FOREIGN KEY (projlead_fk) REFERENCES persnl.employee;
```

- This example changes a table to control the maximum disk space to be allocated:

```
ALTER TABLE persnl.employee ATTRIBUTE MAXEXTENTS 300;
```

- This example shows the steps to drop a foreign key. Suppose you have created two tables, STAFF_M and PROJ_M, and have added foreign key PRI_WK to STAFF_M:

```
CREATE TABLE STAFF_M
           (EMPNUM   CHAR(3) NOT NULL,
            EMPNAME  CHAR(20),
            GRADE    DECIMAL(4),
            CITY     CHAR(15),
            PRI_WK   CHAR(3),
            UNIQUE   (EMPNUM));

        CREATE TABLE PROJ_M
           (PNUM     CHAR(3) NOT NULL,
            PNAME    CHAR(20),
            PTYPE    CHAR(6),
```

```
            BUDGET   DECIMAL(9),
            CITY     CHAR(15),
            MGR   CHAR(3),
            UNIQUE (PNUM),
            FOREIGN KEY (MGR)
            REFERENCES STAFF_M(EMPNUM));

        ALTER TABLE STAFF_M ADD FOREIGN KEY (PRI_WK)
          REFERENCES PROJ_M (PNUM);
```

Suppose further that you now need to drop the foreign key. Use SHOWDDL to obtain the key's system identification:

```
>>showddl staff_m;

CREATE TABLE NIST_EMB_CAT.SUN.STAFF_M
  (
    EMPNUM                              CHAR(3) CHARACTER SET
ISO88591 COLLATE
      DEFAULT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , EMPNAME                             CHAR(20) CHARACTER SET
ISO88591 COLLATE
      DEFAULT DEFAULT NULL
  , GRADE                               DECIMAL(4, 0) DEFAULT NULL
  , CITY                                CHAR(15) CHARACTER SET
ISO88591 COLLATE
      DEFAULT DEFAULT NULL
  , PRI_WK                              CHAR(3) CHARACTER SET
ISO88591 COLLATE
      DEFAULT DEFAULT NULL
  , CONSTRAINT NIST_EMB_CAT.SUN.STAFF_M_452683997_9541 CHECK
      (NIST_EMB_CAT.SUN.STAFF_M.EMPNUM IS NOT NULL) NOT DROPPABLE
  )
  LOCATION \DRP45.$D45101.ZSDBV6VZ.D873HP00
  NAME DRP45_D45101_ZSDBV6VZ_D873HP00
  ;
-- The following index is a system created index --
CREATE UNIQUE INDEX STAFF_M_187893997_9541 ON
NIST_EMB_CAT.SUN.STAFF_M
  (
    EMPNUM ASC
  )
  LOCATION \DRP45.$D45101.ZSDBV6VZ.VTW5HP00
  NAME DRP45_D45101_ZSDBV6VZ_VTW5HP00
  ;
-- The following index is a system created index --
CREATE INDEX STAFF_M_859182618_9541 ON NIST_EMB_CAT.SUN.STAFF_M
  (
    PRI_WK ASC
  )
  LOCATION \DRP45.$D45101.ZSDBV6VZ.SWBSQP00
  NAME DRP45_D45101_ZSDBV6VZ_SWBSQP00
  ;
ALTER TABLE NIST_EMB_CAT.SUN.STAFF_M
  ADD CONSTRAINT NIST_EMB_CAT.SUN.STAFF_M_187893997_9541 UNIQUE
(EMPNUM)
      DROPPABLE ;
ALTER TABLE NIST_EMB_CAT.SUN.STAFF_M
  ADD CONSTRAINT NIST_EMB_CAT.SUN.STAFF_M_859182618_9541 FOREIGN
KEY (PRI_WK)
```

```
        REFERENCES NIST_EMB_CAT.SUN.PROJ_M(PNUM) DROPPABLE ;
--- SQL operation complete.
```

Now that you have the identification, you can drop the foreign key with ALTER TABLE:

```
>>alter table staff_m drop constraint STAFF_M_859182618_9541;
--- SQL operation complete.
```

- The following command renames an existing table `CAT.SCH.T1` to `T2`:

  ```
  ALTER TABLE CAT.SCH.T1 RENAME TO T2;
  ```

  The new ANSI name of the table, T2 is not fully qualified since the table continues to remain in the same catalog and schema as T1. After the table is renamed, the fully qualified name of the table is `CAT.SCH.T2`.

Examples of ALTER TABLE ALTER COLUMN

For a full example of recalibrating an IDENTITY column, see Example of ALTER TABLE ALTER COLUMN..RECALIBRATE on page 2-45.

Create a table with the IDENTITY column

```
CREATE TABLE T1 (surrogate_key LARGEINT GENERATED ALWAYS AS IDENTITY
(START WITH 99 INCREMENT BY 1 MAXVALUE 100 MINVALUE 50 NO CYCLE) NOT NULL, b
INT UNSIGNED NOT NULL,
PRIMARY KEY(surrogate_key) );
```

The third insert will fail with error -8934 as shown here:

```
insert into T1 values(default,1);
--- 1 row(s) inserted.

insert into T1 values(default,2);
--- 1 row(s) inserted.

>>insert into T1 values(default,3);
*** ERROR[8934] The MAXVALUE for the sequence generator has been exceeded.
--- 0 row(s) inserted.
```

Alter the table to allow new MAXVALUE and INCREMENT BY values:

```
ALTER TABLE T1 ALTER COLUMN SURROGATE_KEY SET MAXVALUE 900;
ALTER TABLE T1 ALTER COLUMN SURROGATE_KEY SET INCREMENT BY 2;
insert into T1 values(default,3);
--- 1 row(s) inserted.
select * from T1;
SURROGATE_KEY      B

---------------   ----

99                 1

100                2

102                3
```

Example of ALTER TABLE ALTER COLUMN..RECALIBRATE

1. This example shows how the recalibrate value is adjusted to the INCREMENT BY numbering scheme:

```
CREATE TABLE T127T004 (a LARGEINT
                        GENERATED BY DEFAULT AS IDENTITY
                         (START WITH 9223372036854775804
                          INCREMENT BY 1
                          MAXVALUE 9223372036854775807
                          MINVALUE 100
                          NO CYCLE)
                           NOT NULL NOT DROPPABLE,
                      b     INT UNSIGNED NOT NULL,
                      c     INT UNSIGNED,
                        primary key(a) );

--- SQL operation complete.
alter table t127t004 alter column a recalibrate;

--- SQL operation complete.

-- The CURRENT_VALUE will be 9223372036854775806 after this
-- INSERT
insert into t127t004 values(default,1,1), (default,2,2);

--- 2 row(s) inserted.
select * from t127t004;

A                         B           C

--------------------  ----------  ----------


9223372036854775804        1           1

9223372036854775805        2           2


--- 2 row(s) selected.
alter table t127t004 alter column a set increment by 2;

--- SQL operation complete.

alter table t127t004 alter column a recalibrate;

*** ERROR[1598] The new CURRENT_VALUE, 9223372036854775805
increment by 2 plus 1 to adjust to numbering scheme, for the
IDENTITY column, A, for the table, IC.ICALT.T127T004, will be
greater than the maximum allowed, 9223372036854775807.
```

2. This example illustrates the behavior of Recalibrate to a User-Specified Value with SELECT:

```
CREATE TABLE T004 (colA LARGEINT
                        GENERATED ALWAYS AS IDENTITY
                         (START WITH 700
                          INCREMENT BY 2
                          MAXVALUE 800
                          MINVALUE 100
                          NO CYCLE)
                           NOT NULL NOT DROPPABLE,
                      colB     INT UNSIGNED NOT NULL,
```

```
                      colC     INT UNSIGNED,
                      primary key(colA) );

insert into T004 (colB,colC) values(1,1);
--- 1 row(s) inserted.

-- colA will have value '700';
select * from T004;

COLA                 COLB        COLC

-----------------  ----------  ----------


  700                  1           1


--- 1 row(s) selected.

-- Recalibrate the internal sequence generator
-- using the user-specified value with SELECT.  This
-- will succeed as it is less than the maximum value
-- and greater than the MINVALUE and START WITH values.

-- The recalibrate will reset the CURRENT_VALUE in the SG Table
-- to 710.
alter table t004 alter column colA recalibrate to 710;
--- SQL operation complete.

-- The next number generated for the IDENTITY column colA
-- will be 710, the new recalibrated CURRENT_VALUE

insert into T004 (colB, colC) values(2,2);
--- 1 row(s) inserted.

-- Notice the value 710 for colA.
select * from T004;

COLA         COLB        COLC

---------  --------  ----------


  700          1           1

  710          2           2


--- 2 row(s) selected.

-- The recalibrate performs a SELECT on table T004 to obtain
MAX(colA).
-- Since MAX(colA) is greater than the recalibrate value of 702,
ERROR[1599]
-- is raised.
alter table T004 alter column colA recalibrate to 702;

*** ERROR[1599] The recalibration value is less than or equal to
the current maximum value, 710, of the IDENTITY column, COLA, for
the table, CAT.SCH.T004.

--- SQL operation failed with errors.
```

3. This example illustrates the behavior of “Recalibrate to a User-Specified Value NO SELECT”:

```
CREATE TABLE T004 (colA LARGEINT
                        GENERATED BY DEFAULT AS IDENTITY
                         (START WITH 700
                          INCREMENT BY 2
                          MAXVALUE 800
                          MINVALUE 100
                          NO CYCLE)
                           NOT NULL NOT DROPPABLE,
                      colB     INT UNSIGNED NOT NULL,
                      colC     INT UNSIGNED,
                      primary key(colA) );

insert into T004 (colB,colC) values(1,1);
--- 1 row(s) inserted.

-- colA will have value '700';
select * from T004;

COLA         COLB        COLC

---------   ---------  ----------

 700           1           1

--- 1 row(s) selected.

-- Recalibrate the internal sequence generator
-- using the user-specified value with SELECT.  This
-- will succeed as it is less than the maximum value
-- and greater than the MINVALUE and START WITH values.
-- The recalibrate will reset the CURRENT_VALUE in the SG Table
-- to 710.
alter table t004 alter column colA recalibrate to 710;
--- SQL operation complete.

-- The recalibrate does not perform a SELECT on table T004 to
obtain MAX(colA).
-- The recalibrate to value of 702 is a pure override. No error is
raised.
alter table T004 alter column colA recalibrate to 702;
--- SQL operation complete.
```

Notice the plus 1 that is done to calculate the new `CURRRENT_VALUE`; that is because adding the INCREMENT BY 2 to 9223372036854775805 will result in an odd number, whereas the sequence generator is defined to generate even values. Hence the plus 1 is done to be consistent to the numbering scheme.

ALTER TRIGGER Statement

Considerations for ALTER TRIGGER

The ALTER TRIGGER statement is used to enable or disable triggers, individually or by SQL/MX table.

```
ALTER TRIGGER { ENABLE trigger-name        |
                ENABLE ALL OF table-name |
                DISABLE trigger-name       |
                DISABLE ALL OF table-name};
```

Syntax Description of ALTER TRIGGER

trigger-name

specifies the ANSI logical name of the trigger to be altered, of the form:

[[*catalog-name*.]*schema-name*.]*trigger-name*

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

table-name

specifies the ANSI logical name of the table that this trigger is defined on, of the form:

[[*catalog-name*.]*schema-name*.]*table-name*

where each part of the name is a valid SQL identifier with a maximum of 128 characters.

Considerations for ALTER TRIGGER

ENABLE ALL enables all triggers defined on *table-name*.

DISABLE ALL disables all triggers defined on *table-name*.

Authorization and Availability Requirements

To alter a trigger, you must own its schema or be the super ID or object owner. Only the super ID can use ALTER TRIGGER DISABLE ALL or ALTER TRIGGER ENABLE ALL.

ALTER VIEW Statement

This section describes the ALTER VIEW statement and examples illustrating the statement.

Considerations for ALTER VIEW
Example of ALTER VIEW

The ALTER VIEW statement alters a view by performing the following actions:

- Renaming an object within a schema
- Modifying a view by changing one or more view file attributes
- Enabling or disabling Similarity Checks. This is supported from SQL/MX Release 3.2

```
ALTER VIEW name alter-action

alter-action is: {rename-action | similarity-check-action}

rename-action is:
RENAME TO new-view-name

similarity-check-action is:
[{ENABLE | DISABLE} SIMILARITY CHECK]
```

name

specifies the name of the view to alter.

RENAME TO *new-view-name*

changes the logical name of the view.

new-view-name

specifies the new ANSI name of the view. The new ANSI name of the view cannot be qualified; the renamed view remains in the current catalog and schema.

Note.

- Renaming a view changes the text of referencing views, RI constraints, and triggers to refer to the new name.
- RENAME TO changes the redefinition timestamp of the affected view. However, the tables referenced by the view are not affected.

similarity-check-option

`ENABLE SIMILARITY CHECK`

enables Similarity Check for the view.

`DISABLE SIMILARITY CHECK`

disables Similarity Check for the view.

Considerations for ALTER VIEW

Authorization and Availability Requirements

To alter a view, you must own its schema or be the super ID or object owner.

Renaming a View

You can use the rename option to change the name of a view. The following are the prerequisites for renaming a view:

- The view and its file label must be available.
- The new ANSI name must not already exist within the schema.

Similarity Check

- For the Similarity Check setting to become effective after issuing an ALTER VIEW statement, you must recompile the applications using the view.
- An error is returned if an attempt is made to ENABLE SIMILARITY CHECK on a view for which Similarity Check is already enabled or DISABLE SIMILARITY CHECK on a view for which Similarity Check is already disabled.

Effects on TMF

Rename

For information on effects of the rename option on TMF, see Effects on TMF on page 2-40.

Example of ALTER VIEW

- The following command renames the view, `CAT.SCH.V1` to `V2`:

```
ALTER VIEW CAT.SCH.V1 RENAME TO V2;
```

The new view, V2 continues to remain in the same catalog and schema as `V1.`

- The following command enables Similarity Check on the view:

```
ALTER VIEW CAT.SCH.V1 ENABLE SIMILARITY CHECK;
```

- The following command disables Similarity Check on the view:

```
ALTER VIEW CAT.SCH.V1 DISABLE SIMILARITY CHECK;
```

BEGIN WORK Statement

The BEGIN WORK statement enables you to start a transaction explicitly—where the transaction consists of the set of operations defined by the sequence of SQL statements that begins immediately after BEGIN WORK and ends with the next COMMIT or ROLLBACK statement. See Transaction Management on page 1-13.

BEGIN WORK is an SQL/MX extension.

```
BEGIN WORK
```

Considerations for BEGIN WORK

Effect on Audited Tables

A user-defined transaction groups together a set of operations on audited tables so that changes made by the operations can be committed (with the COMMIT statement) or rolled back (with the ROLLBACK statement) as a unit. That is, the sequence of SQL statements that make up the transaction either completely executes or has no effect.

Effect on Nonaudited Tables

Transactions do not protect nonaudited tables. The BEGIN WORK statement has no effect on nonaudited tables.

MXCI Examples of BEGIN WORK

- Group three separate statements—two INSERT statements and an UPDATE statement—that update the database within a single transaction:

```
--- This statement initiates a transaction.
BEGIN WORK;
--- SQL operation complete.

INSERT INTO sales.orders VALUES (125, DATE '1998-03-23',
  DATE '1998-03-30', 75, 7654);
--- 1 row(s) inserted.

INSERT INTO sales.odetail VALUES (125, 4102, 25000, 2);
--- 1 row(s) inserted.

UPDATE invent.partloc SET qty_on_hand = qty_on_hand - 2
  WHERE partnum = 4102 AND loc_code = 'G45';
--- 1 row(s) updated.

--- This statement ends a transaction.
COMMIT WORK;
--- SQL operation complete.
```

C Examples of BEGIN WORK

- Begin a transaction, execute an UPDATE statement, and test SQLSTATE. If the UPDATE is successful, the database changes are committed. Otherwise, the database changes are rolled back:

```
...
CHAR SQLSTATE_OK[6] = "00000";
EXEC SQL BEGIN DECLARE SECTION;
  CHAR SQLSTATE[6];
  ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL BEGIN WORK;            /* Start a transaction. */
...
EXEC SQL UPDATE ... ;           /* Change the database. */
...
if (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
   EXEC SQL COMMIT WORK;         /* Commit the changes. */
else
   EXEC SQL ROLLBACK WORK;    /* Roll back the changes. */
```

COBOL Examples of BEGIN WORK

- Begin a transaction, execute an UPDATE statement, and test SQLSTATE. If the UPDATE is successful, the database changes are committed. Otherwise, the database changes are rolled back:

```
 ...
  01 SQLSTATE-OK     PIC X(5) VALUE "00000".
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 SQLSTATE        PIC X(5).
     ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
* Start a transaction.
     EXEC SQL BEGIN WORK END-EXEC.
     ...
* Change the database.
     EXEC SQL UPDATE ... END-EXEC.
     ...
* Commit or roll back the changes.
     IF SQLSTATE = SQLSTATE-OK
       EXEC SQL COMMIT WORK END-EXEC.
     ELSE
       EXEC SQL ROLLBACK WORK END-EXEC.
```

CALL Statement

[Considerations for CALL](#)
[Examples of CALL](#)

The CALL statement invokes a stored procedure in Java (SPJ) in NonStop SQL/MX. To develop, deploy, and manage SPJs in SQL/MX, see the *SQL/MX Guide to Stored Procedures in Java*.

```
CALL procedure-ref ([argument-list])

procedure-ref is:
   [[catalog-name.]schema-name.]procedure-name

argument-list is:
   SQL-expression[{, SQL-expression}...]
```

procedure-ref

specifies an ANSI logical name of the form:

`[[catalog-name.]schema-name.]procedure-name`

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

If you do not fully qualify the procedure name, NonStop SQL/MX qualifies it according to the current settings of CATALOG and SCHEMA. If you set the NAMETYPE attribute to NSK instead of ANSI and you do not fully qualify the procedure name, NonStop SQL/MX returns an error. For more information on the CATALOG, SCHEMA, and NAMETYPE attributes, see the System Defaults Table on page 10-37.

argument-list

accepts arguments for IN, INOUT, or OUT parameters. The arguments consist of SQL expressions, including host variables or dynamic parameters, separated by commas:

`SQL-expression[{, SQL-expression}...]`

Each expression must evaluate to a value of one of these data types:

- Character value
- Date-time value
- Numeric value

Interval value expressions are disallowed in SPJs. For more information, see Input Parameter Arguments on page 2-55 and Output Parameter Arguments on page 2-55.

Considerations for CALL

Usage Restrictions

You can use the CALL statement only as a stand-alone SQL statement in applications or interfaces that call NonStop SQL/MX. You cannot use the CALL statement inside a compound statement, or with rowsets. Starting with SQL/MX Release 3.2, stored procedures can be called from triggers.

Required Privileges

To execute the CALL statement, you must have EXECUTE privilege on the procedure. For more information, see the GRANT EXECUTE Statement on page 2-246.

Input Parameter Arguments

You pass data to an SPJ by using IN or INOUT parameters. For an IN parameter argument, use one of these SQL expressions:

- Literal
- SQL function (including CASE and CAST expressions)
- Arithmetic or concatenation operation
- Scalar subquery
- Host variable (for example, `:hostvar`)
- Dynamic parameter (for example, `?` or `?param`)

For more information, see Expressions on page 6-41.

For an INOUT parameter argument, you can use only a host variable or dynamic parameter.

Output Parameter Arguments

An SPJ returns values in OUT and INOUT parameters. Output parameter arguments must be either host variables in a static CALL statement (for example, `:hostvar`) or dynamic parameters in a dynamic CALL statement (for example, `?` or `?param`). Each calling application defines the semantics of the OUT and INOUT parameters in its environment. For more information, see the *SQL/MX Guide to Stored Procedures in Java*.

Data Conversion of Parameter Arguments

NonStop SQL/MX performs an implicit data conversion when the data type of a parameter argument is compatible with but does not match the formal data type of the stored procedure. For stored procedure input values, the conversion is from the actual argument value to the formal parameter type. For stored procedure output values, the conversion is from the actual output value, which has the data type of the formal parameter, to the declared type of the host variable or dynamic parameter.

Null Input and Output

You can pass a null value as input to or output from an SPJ, provided that the corresponding Java data type of the parameter supports nulls. If a null is input or output for a parameter that does not support nulls, NonStop SQL/MX returns an error. For more information on handling null input and output, see the *SQL/MX Guide to Stored Procedures in Java*.

Transaction Semantics

The CALL statement automatically initiates a transaction if there is no active transaction. However, the failure of a CALL statement does not always automatically abort the transaction. For more information, see the *SQL/MX Guide to Stored Procedures in Java*.

Examples of CALL

- In MXCI, invoke an SPJ named MONTHLYORDERS, which has one IN parameter represented by a literal and one OUT parameter represented by a dynamic parameter ?:

  ```
  CALL samdbcat.sales.monthlyorders(3,?);
  ```

- From an embedded SQL program in C, invoke an SPJ named MONTHLYORDERS, which has an OUT parameter represented by a host variable:

  ```
  EXEC SQL CALL samdbcat.sales.monthlyorders(3,:orders);
  ```

For more examples, see the *SQL/MX Guide to Stored Procedures in Java*.

COMMIT WORK Statement

The COMMIT WORK statement commits any changes to audited objects made during the current transaction, releases all locks on audited objects held by the transaction, and ends the transaction. See Transaction Management on page 1-13.

```
COMMIT [WORK]
```

WORK is an optional keyword that has no effect.

COMMIT WORK has no effect outside of an active transaction.

Embed COMMIT WORK closes all open cursors in the application, because cursors do not span transaction boundaries. You cannot fetch with a cursor after a transaction ends without reopening the cursor. ■

Considerations for COMMIT WORK

Begin and End a Transaction

BEGIN WORK starts a transaction. COMMIT WORK or ROLLBACK WORK ends a transaction. Committing a transaction without verifying the successful completion of DML statements within the transaction boundary can create inconsistent data. Therefore, you must verify the successful completion of the DML statement within the transaction boundary.

Effect of Constraints

When COMMIT WORK is executed, all active constraints are checked, and if any constraint is not satisfied, changes made to the database by the current transaction are canceled—that is, work done by the current transaction is rolled back. If all constraints are satisfied, all changes made by the current transaction become permanent.

MXCI Examples of COMMIT WORK

- Suppose that your application adds information to the inventory. You have received 24 terminals from a new supplier and want to add the supplier and update the quantity on hand. The part number for the terminals is 5100, and the supplier is assigned supplier number 17. The cost of each terminal is $800.

The transaction must add the order for terminals to PARTSUPP, add the supplier to the SUPPLIER table, and update QTY_ON_HAND in PARTLOC. After the INSERT and UPDATE statements execute successfully, you commit the transaction, as shown, within MXCI:

```
-- This statement initiates a transaction.
BEGIN WORK;
--- SQL operation complete.

-- This statement inserts a new entry into PARTSUPP.
INSERT INTO invent.partsupp
VALUES (5100, 17, 800.00, 24);
--- 1 row(s) inserted.

-- This statement inserts a new entry into SUPPLIER.
INSERT INTO invent.supplier
VALUES (17, 'Super Peripherals','751 Sanborn Way',
  'Santa Rosa', 'California', '95405');
--- 1 row(s) inserted.

-- This statement updates the quantity in PARTLOC.
UPDATE invent.partloc
SET qty_on_hand = qty_on_hand + 24
WHERE partnum = 5100 AND loc_code = 'G43';
--- 1 row(s) updated.

-- This statement ends a transaction.
COMMIT WORK;
--- SQL operation complete.
```

C Examples of COMMIT WORK

- Begin a transaction, execute an UPDATE statement, and test SQLSTATE. If the UPDATE is successful, the database changes are committed. Otherwise, the database changes are rolled back.

```
...
CHAR SQLSTATE_OK[6] = "00000";
EXEC SQL BEGIN DECLARE SECTION;
  CHAR SQLSTATE[6];
  ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL BEGIN WORK;            /* Start a transaction. */
...
EXEC SQL UPDATE ... ;           /* Change the database. */
...
if (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
   EXEC SQL COMMIT WORK;         /* Commit the changes. */
else
   EXEC SQL ROLLBACK WORK;    /* Roll back the changes. */
```

COBOL Examples of COMMIT WORK

- Begin a transaction, execute an UPDATE statement, and test SQLSTATE. If the UPDATE is successful, the database changes are committed. Otherwise, the database changes are rolled back.

```
     ...
  01 SQLSTATE-OK    PIC X(5) VALUE "00000".
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 SQLSTATE       PIC X(5).
     ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
* Start a transaction.
     EXEC SQL BEGIN WORK END-EXEC.
     ...
* Change the database.
     EXEC SQL UPDATE ... END-EXEC.
     ...
* Commit or roll back the changes.
    IF SQLSTATE = SQLSTATE-OK
       EXEC SQL COMMIT WORK END-EXEC.
    ELSE
       EXEC SQL ROLLBACK WORK END-EXEC.
```

CONTROL QUERY DEFAULT Statement

Considerations for CONTROL QUERY DEFAULT
Examples of CONTROL QUERY DEFAULT

The CONTROL QUERY DEFAULT statement changes the system-level default settings for the current process. Execution of this statement does not change the contents of the SYSTEM_DEFAULTS table. See System Defaults Table on page 10-37.

CONTROL QUERY DEFAULT is an SQL/MX extension.

```
CONTROL QUERY DEFAULT control-default-option

control-default-option is:
     attribute {'attr-value' | RESET}
   | * RESET
```

attribute

is a character string that represents an SQL/MX attribute name and corresponds to the ATTRIBUTE column of the SYSTEM_DEFAULTS table. For descriptions of these attributes, see Default Attributes on page 10-39.

attr-value

is a character string that specifies an SQL/MX attribute value and corresponds to the ATTR_VALUE column of the SYSTEM_DEFAULTS table. You must specify it as a quoted string—even if the value is a number.

RESET

specifies that the attribute that you set by using a CONTROL QUERY DEFAULT statement in the current session is to be reset to the value or values in effect at the start of the current session.

*

specifies all attributes are to be reset.

Considerations for CONTROL QUERY DEFAULT

Scope of CONTROL QUERY DEFAULT

The result of the execution of a CONTROL QUERY DEFAULT statement stays in effect until the current process terminates or until the execution of another statement for the same *attribute* overrides it. For a detailed list of the precedence of default settings, see System Defaults Table on page 10-37.

Relationship to CONTROL TABLE

Use the CONTROL QUERY DEFAULT statement to override system-level default settings for various attributes for all tables in the current process. Use the CONTROL TABLE statement to override system-level default settings for the TABLELOCK, TIMEOUT, and SIMILARITY_CHECK attributes per table or for all tables in the current process. For example, the statement `CONTROL QUERY DEFAULT TIMEOUT '3000'` has the same effect as the statement `CONTROL TABLE * TIMEOUT '3000'`. See CONTROL TABLE Statement on page 2-74.

The CONTROL TABLE statement has precedence over the CONTROL QUERY DEFAULT statement. For a detailed list of the precedence of default settings, see System Defaults Table on page 10-37.

Examples of CONTROL QUERY DEFAULT

- Change the ISOLATION_LEVEL attribute for the current process:

  ```
  CONTROL QUERY DEFAULT ISOLATION_LEVEL 'READ UNCOMMITTED';
  ```

- Change the static TIMEOUT attribute for the current process:

  ```
  CONTROL QUERY DEFAULT TIMEOUT '3000';
  ```

  The value 3000 is in hundredths of seconds, which is equivalent to 30 seconds.

- Reset the TIMEOUT attribute to its initial value in the current process:

  ```
  CONTROL QUERY DEFAULT TIMEOUT RESET;
  ```

- Specify which volumes to use for scratch disks and which volume is preferred:

  ```
  CONTROL QUERY DEFAULT SCRATCH_DISKS
    '$data01, $data02, \tstnode.$scr';
  ```

  ```
  CONTROL QUERY DEFAULT SCRATCH_DISKS_PREFERRED '$data02';
  ```

CONTROL QUERY SHAPE Statement

Considerations for CONTROL QUERY SHAPE
Examples of CONTROL QUERY SHAPE

The CONTROL QUERY SHAPE statement forces access plans. You can generate the result of the EXPLAIN function for a prepared DML statement, and then modify the operator tree for the statement's access plan, by using CONTROL QUERY SHAPE. Both the EXPLAIN function and the CONTROL QUERY SHAPE statement use similar identifiers for the nodes of an operator tree.

CONTROL QUERY SHAPE is an SQL/MX extension:

```
CONTROL QUERY SHAPE [ query-shape-options ] shape

query-shape-options  is:

     {IMPLICIT { SORT | EXCHANGE | EXCHANGE_AND_SORT }}

shape is:
     {CUT | ANYTHING | OFF}
   | TUPLE
   | join-shape(shape,shape [,{TYPE1 | TYPE2}]
     [,num-of-esps]) | INDEXJOIN
   | exchange-shape(shape [,num-of-esps])
   | {SCAN | FILE_SCAN | INDEX_SCAN} [(scan-option,...)]
   | UNION(shape,shape)
   | MultiUnion(shape,[shape])
   | GROUPBY(shape)
   | SORT_GROUPBY(shape)
   | HASH_GROUPBY(shape)
   | SHORTCUT_GROUPBY(shape)
   | EXPR(shape)
   | SORT(shape)
   | MATERIALIZE(shape)
   | [{PACK | UNPACK }] (shape)

join-shape is:
     JOIN
   | NESTED_JOIN
   | MERGE_JOIN
   | HASH_JOIN
   | HYBRID_HASH_JOIN
   | ORDERED_HASH_JOIN

exchange-shape is:
     EXCHANGE
   | PARTITION_ACCESS
   | SPLIT_TOP_PA
   | ESP_EXCHANGE
```

```
scan-option is:
     TABLE table
   | PATH {access-path | ANY}
   | BLOCKS_PER_ACCESS value
   | MDAM mdam-option,...
   | MDAM_COLUMNS mdam-columns-option,...

mdam-option is:
     OFF
   | SYSTEM
   | FORCED

mdam-columns-option is:
     n
   | SYSTEM
   | ALL
   | (density,...)
   | SYSTEM (density,...)
   | ALL (density,...)

density is:
     SPARSE
   | DENSE
   | SYSTEM
```

query-shape-options IMPLICIT
{ SORT | EXCHANGE | EXCHANGE_AND_SORT }

makes SORT, EXCHANGE, or ENFORCER nodes in CONTROL QUERY SHAPE optional.

IMPLICIT SORT

indicates that the optimizer can add sort nodes at any location between the nodes forced in the shape statements. The shape statement can still contain additional sort nodes, and the compiler will enforce such nodes. Note that WITHOUT SORT does not mean that the statement or the generated plans are free of sort nodes.

IMPLICIT EXCHANGE

indicates that the optimizer can add exchange nodes at any location between the nodes forced in the shape statements. The shape statement can still contain additional exchange nodes, and the compiler will enforce such nodes. Note that WITHOUT EXCHANGE does not mean that the statement or the generated plans are free of exchange nodes.

IMPLICIT EXCHANGE_AND_SORT

indicates that the optimizer can add enforcer nodes (exchanges or sorts) at any location between the nodes forced in the shape statements. The shape statement can still contain additional enforcer nodes, and the compiler will enforce such nodes. Note that WITHOUT ENFORCERS does not mean that the statement or the generated plans are free of enforcer nodes.

shape

specifies the operator tree for an access plan for the next query to be executed within the current session. *shape* defines this operator tree recursively by using a LISP-like expression to represent the nodes of the tree.

The identifiers for the nodes specified by *shape*:

CUT ANYTHING OFF	Replaces a node in operator tree. `CUT` (or `ANYTHING` or `OFF`) represents the point (node) at which you want to cut off the remaining part of the operator tree.
TUPLE	Replaces a single VALUES node in operator tree.
UNION	Replaces a UNION node in operator tree.

MultiUnion	MultiUnion is the first `n-ary` operator that is supported in a Control Query Shape (CQS). MultiUnion is applicable only for the UNION ALL operator and not for the ANSI UNION operator.
	
	You can specify a shape for MultiUnion children in one of the following ways:
	
	1. Allow a single wild card shape for all the children. All children are assumed to look alike.
	2. Allow multiple wild card shape for children. A group of children conform to one set of shape, the rest to another.
	3. Specify a shape for each MultiUnion child completely.
	
	If the third alternative is implemented—that is if a MultiUnion has N children, all the N children must be specified in the shape. The children must not be excluded. Any attempt to match a MultiUnion with different arity will result in a violation of the shape specification. However, the generic wild-card operator `cut` can be used as a replacement for any of the children of the MultiUnion. For example:
	
	`MultiUnion(cut,`
	`Nested_Join(cut,cut),cut);`

GROUPBY	Replaces a GROUP BY or AGGREGATE (set) operation node in operator tree.
SORT_GROUPBY	Replaces a SORT GROUP BY or AGGREGATE (set) operation node in operator tree.
HASH_GROUPBY(cut) or HG (cut)	Replaces a HASH GROUP BY node in operator tree.

SHORTCUT_GROUPBY	Replaces an AGGREGATE (set) operation node in operator tree.
EXPR	Replaces a map value IDs node in operator tree.
SORT	Replaces a SORT node in operator tree.
EXCHANGE	Replaces any of three nodes—PARTITION ACCESS, SPLIT TOP / PARTITION ACCESS, and REPARTITION—in operator tree.
PARTITION_ACCESS	Replaces a PARTITION ACCESS node in operator tree.
SPLIT_TOP_PA	Replaces a SPLIT TOP / PARTITION ACCESS node pair in operator tree.
ESP_EXCHANGE	Replaces an ESP_EXCHANGE node in operator tree.
MATERIALIZE	Replaces a materialized temporary table node in operator tree.
PACK	Optionally replaces a PACK node in operator tree. This node relates to rowsets.
UNPACK	Optionally replaces an UNPACK node in operator tree. This node relates to rowsets.
SCAN	Replaces a SCAN node in operator tree.
FILE_SCAN	Replaces a SCAN node in operator tree.
INDEX_SCAN	Replaces a SCAN node in operator tree.
JOIN	Replaces a JOIN node in operator tree.
NESTED_JOIN	Replaces a NESTED_JOIN in operator tree.
MERGE_JOIN	Replaces a MERGE_JOIN in operator tree.
HASH_JOIN (cut,cut) or HJ (cut,cut)	Replaces a HYBRID_HASH_JOIN or ORDERED_HASH_JOIN in operator tree.
HYBRID_HASH_JOIN (cut,cut) or HHJ (cut,cut)	Replaces a HYBRID_HASH_JOIN in operator tree.
ORDERED_HASH_JOIN (cut,cut) or OHJ (cut,cut)	Replaces an ORDERED_HASH_JOIN in operator tree.

```
join-shape (shape,shape [,{TYPE1 | TYPE2}]
    [,num-of-esps]) | INDEXJOIN
```

specifies the join type and the number of ESPs that you can use with JOIN, NESTED_JOIN, MERGE_JOIN, HASH_JOIN, HYBRID_HASH_JOIN, or ORDERED_HASH_JOIN as follows:

Note. Regarding the ORDERED_HASH_JOIN and HYBRID_HASH_JOIN join shapes, when ORDERED_HASH_JOIN is set, the optimizer forces the hash table (the inner/right table in the join) into memory, so the order of the left/outer table is preserved. However, if the inner table is very large (millions of rows), NonStop SQL/MX can run out of read-only memory, so NonStop virtual memory would be used (flush pages in and out of disk, and so on.) This virtual memory is slower than the HYBRID_HASH_JOIN mechanism. If you want to guarantee the order of the left/outer table, you can use ORDERED_HASH_JOIN. Consider the potential performance implication (depends on size of the right table, size of memory, workload, and so on). You might want to explicitly use HYBRID_HASH_JOIN to avoid the performance issue. However, if you specify HASH_JOIN (cut,cut), the optimizer will present a plan with either HYBRID_HASH_JOIN or ORDERED_HASH_JOIN as shown in the SHOWSHAPE of the resulting plan.

`TYPE1`

specifies a parallel join with matching partitions.

`TYPE2`

specifies a join with parallel access to the inner table.

`num-of-esps`

specifies the number of ESPs

`INDEXJOIN`

specifies that the optimizer should use the index and base table to create a scan, if possible.

`{SCAN | FILE_SCAN | INDEX_SCAN} [(`*`scan-option`*`,...)]`

specifies the options that you can use with SCAN, FILE_SCAN, or INDEX_SCAN, as follows:

`TABLE` *`table`*

specifies a table or correlation name. *`table`* is a character string literal—for example, '`t1`'.

`PATH {`*`access-path`*` | ANY}`

`access-path` specifies an index name or the table name *`table`*. It is a character string literal—for example, `'ix1'` or `'t1'`. Use this option to force the scan to be on that particular index or base table. The specified *`access-path`* must cover all of the predicates defined on the table columns for a plan to be generated.

`access-path` for an MP index can contain only the last part of the filename. For example:

```
control query shape partition_access(scan(path
'INAME',forward, blocks_per_access 1 , mdam off));
```

`ANY` forces the scan to be on any available access path—an index or the base table. In this case, the optimizer chooses the access path on the basis of cost.

If you do not use the `PATH` option, the default is `ANY`—except if the MultiDimensional Access Method (MDAM) is forced, which changes the default to the base table.

`BLOCKS_PER_ACCESS `*`value`*

specifies for the HP NonStop Data Access Manager (DAM) how many blocks DAM can read ahead. DAM limits the number of read-ahead blocks to 14 at a time. Use this token to control the read ahead in a mixed workload environment to minimize its effect on other transactions. A setting of 1 (one) prevents read ahead.

`MDAM `*`mdam-option`*`,...`

specifies MDAM options:

`OFF`

disables the MDAM option for this scan.

`SYSTEM`

specifies that the system determines whether to use MDAM on the basis of cost.

`FORCED`

forces the use of MDAM for the scan. Using any MDAM_COLUMNS option implies this option.

`MDAM_COLUMNS `*`mdam-columns-option`*`,...`

specifies these MDAM_COLUMNS options:

n

specifies *n* key columns to be used by MDAM. For example, if MDAM is using key columns specified by an index defined on a table, *n* specifies that the first *n* columns listed in the index are used. The enumeration algorithm for each column is determined by the system.

This option is the same as ALL if *n* exceeds the total number of columns that MDAM can use.

`SYSTEM`

specifies that the system determines the number of key columns to be used by MDAM on the basis of cost. The enumeration algorithm for each column is determined by the system.

`ALL`

specifies that MDAM use all key columns. The enumeration algorithm for each column is determined by the system.

`(density,...)`

specifies a list of enumeration algorithms. The *density* list forces which algorithms MDAM uses for a set of key columns. If there are *n* algorithms in the list, MDAM uses the first *n* key columns. If *n* exceeds the total number of key columns, MDAM uses only the algorithms for the number of available columns.

density is defined as:

`SPARSE`

specifies the use of the sparse algorithm for the column.

`DENSE`

specifies the use of the adaptive dense algorithm for the column.

`SYSTEM`

specifies that the system determines the enumeration algorithm for the column on the basis of cost.

`SYSTEM (density,...)`

forces an enumeration algorithm to be used by MDAM for each key column as determined by the *density* list. The system determines enumeration algorithms for columns beyond those specified by the *density* list.

`ALL (density,...)`

forces an enumeration algorithm to be used by MDAM for each key column as determined by the *density* list. All of the remaining columns are used by MDAM, and their enumeration algorithms are determined by the system.

For more information, see the *SQL/MX Query Guide*.

Considerations for CONTROL QUERY SHAPE

Scope of CONTROL QUERY SHAPE

The result of the execution of a CONTROL QUERY SHAPE statement stays in effect until the end of the current MXCI session or until the execution of another CONTROL QUERY SHAPE statement overrides it.

If you do not execute CONTROL QUERY SHAPE OFF immediately after the execution of the query with the forced plan, the next query might not fit the forced plan in effect and will return an optimizer error and not compile. The error returned by NonStop SQL/MX has the form:

```
*** ERROR[2105] This query could not be compiled because of
incompatible Control Query Shape (CQS) specifications.
Inspect the CQS in effect.

*** ERROR[8822] The statement was not prepared.
```

The result of the execution of a CONTROL QUERY SHAPE statement does not affect the execution of CONTROL statements, the LOCK and UNLOCK statements, and transaction statements.

Examples of CONTROL QUERY SHAPE

- Switch `OFF` the forced plan for the current MXCI session:

```
CONTROL QUERY SHAPE OFF;
```

- Use `CUT` to shape a partial operator tree:

```
CONTROL QUERY SHAPE JOIN (CUT,UNION(CUT,SCAN));
```

- This example shapes the EXPLAIN operator tree by using the CONTROL QUERY SHAPE statement. Suppose that you have this query:

```
SELECT * FROM employee, dept
  WHERE employee.deptnum = dept.deptnum
    AND employee.last_name = 'SMITH';

Employee/Number  First Name    Last Name      Dept/Num  ...
---------------  ------------  -------------  --------  ...
             89  PETER         SMITH              3300  ...

--- 1 row(s) selected.
```

By using the SET SHOWSHAPE command, you can inspect the default plan generated by the optimizer:

```
SET SHOWSHAPE ON;

SELECT * FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPTNUM = DEPT.DEPTNUM
  AND EMPLOYEE.LAST_NAME = 'SMITH';

control query shape merge_join(sort(
partition_access(scan('EMPLOYEE', forward,
blocks_per_access 1, mdam off))),
partition_access(scan('DEPT', forward,
blocks_per_access 3, mdam off)));
```

Instead of using the default MERGE_JOIN and SORT for this query, you can shape the EXPLAIN operator tree by using this NESTED_JOIN replacement:

```
CONTROL QUERY SHAPE
  NESTED_JOIN (PARTITION_ACCESS(SCAN),
               PARTITION_ACCESS(SCAN('DEPT')));

SET SHOWSHAPE ON;

SELECT * FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPTNUM = DEPT.DEPTNUM
  AND EMPLOYEE.LAST_NAME = 'SMITH';

control query shape nested_join(
partition_access(scan('EMPLOYEE', forward,
blocks_per_access 1, mdam off)),
partition_access(scan('DEPT', forward,
blocks_per_access 1 , mdam off)));

Employee/Number  First Name    Last Name      Dept/Num  ...
---------------  ------------  -------------- --------  ...
             89  PETER         SMITH              3300  ...

--- 1 row(s) selected.
```

The second CONTROL QUERY SHAPE statement is displayed by the SET SHOWSHAPE ON statement. Notice that, because you specified DEPT, you do not have to specify EMPLOYEE. The system uses the other table in the join as the default table name.

- Suppose that you have a table T1 consisting of columns A, B, C, D, E, and F with a primary key defined as columns A and B. Suppose further that an index IT1 is defined as columns C, D, E, and F of table T1. These examples illustrate some of the scan options you can specify for table T1:
  - Scan table T1. You want the system to choose whether to use an index or base table, in addition to the other scan options.

    ```
    SCAN (TABLE 'T1')
    ```

or

```
SCAN (TABLE 'T1', PATH ANY)
```

- Scan table T1 using the base table. You want the system to choose whether to use MDAM.

```
SCAN (TABLE 'T1', PATH 'T1')
```

- Scan table T1 using the index IT1. You want the system to choose whether to use MDAM.

```
SCAN (TABLE 'T1', PATH 'IT1')
```

- Scan table T1 using the index IT1. You want to disable MDAM.

```
SCAN (TABLE 'T1', PATH 'IT1', MDAM OFF)
```

- Scan table T1 using MDAM on index IT1. You want the system to choose the number of MDAM key columns and the enumeration algorithm for each column.

```
SCAN (TABLE 'T1', PATH 'IT1', MDAM FORCED)
```

- Scan table T1 using MDAM on columns C, D, and E of index IT1. You want the system to choose the enumeration algorithm for each column.

```
SCAN (TABLE 'T1', PATH 'IT1', MDAM FORCED,MDAM_COLUMNS 3)
```

or

```
SCAN (TABLE 'T1', PATH 'IT1', MDAM_COLUMNS 3)
```

- Scan table T1 using MDAM on columns C, D, and E of index IT1. The enumeration algorithms for columns C and E are adaptive dense and sparse respectively. You want the system to choose the enumeration algorithm for column D.

```
SCAN (TABLE 'T1', PATH 'IT1',
      MDAM_COLUMNS (DENSE,SYSTEM,SPARSE))
```

- Suppose that you are trying to refine a shape. You want to push a GROUPBY operator down over a hybrid hash join between two tables, A and B. Until the shape is final, you are not concerned with sorts for a possible SORT GROUPBY or a final ordering, or about EXCHANGE nodes. You might add those nodes back after you have finalized the shape:

```
CONTROL QUERY SHAPE WITHOUT ENFORCERS
hybrid_hash_join(groupby(scan('A')), scan('B'));

optimizer/opt.cpp
optimizer/opt.h
optimizer/OptPhysRelExpr.cpp
optimizer/PhyProp.cpp
optimizer/RelControl.h
optimizer/RelExpr.cpp
parser/ParKeyWords.cpp
```

```
parser/SqlParser.y
regress/compGeneral/EXPECTED018
regress/compGeneral/TEST018
```

- This example shows how a scan can be forced to use an index:

```
>>prepare xx from
+>select * from part where p_partkey = (select
max(ps_partkey) from partsupp);

--- SQL command prepared.
>>explain options 'f' xx;

LC    RC   OP  OPERATOR            OPT  DESCRIPTION        CARD
--    --   --  -------------       ---  -----------        ----------
7     .    8   root                                        1.00E+000
4     6    7   nested_join                                 1.00E+000
5     .    6   partition_access                            1.00E+000
.     .    5   file_scan_unique    fr   PART (s)           1.00E+000
3     .    4   partition_access                            1.00E+000
2     .    3   shortcut_scalar_aggr                        1.00E+000
1     .    2   firstn                                      1.00E+000
.     .    1   index_scan               PSX2 (s)           1.00E+002
--- SQL operation complete.
```

Force the scan to go through the index, which could be more efficient:

```
>>control query shape
nested_join(shortcut_groupby(split_top_pa(scan(path
'TPCDF.SF100f.PSX2'),
+>group , 72)), nested_join(exchange(scan(path
'TPCDF.SF100F.PX1')),
+>exchange(scan(path 'TPCDF.SF100F.PART')), INDEXJOIN));


--- SQL operation complete.
```

Prepare the statement using the forced scan:

```
>>prepare xx from
+>select * from part where p_partkey = (select
max(ps_partkey) from partsupp);

--- SQL command prepared.
>>explain options 'f' xx;

LC    RC   OP  OPERATOR            OPT  DESCRIPTION        CARD
--    --   --  -------------       ---  -----------        ----------
13    .    14  root                                         1.00E+000
5     12   13  nested_join                                  1.00E+000
8     11   12  nested_join                                  1.00E+000
10    .    11  split_top                  1:72(logph)       1.00E+000
9     .    10  partition_access                             1.00E+000
.     .    9   file_scan_unique    fr     PART (s)          1.00E+000
7     .    8   split_top                  1:72(logph)       6.66E+006
6     .    7   partition_access                             6.66E+006
.     .    6   index_scan          fr     PX1 (s)           6.66E+006
```

```
4     .    5   shortcut_scalar_aggr                            1.00E+000
3     .    4   firstn                                          1.00E+000
2     .    3   split_top                 1:72(logph)           8.00E+007
1     .    2   partition_access                                8.00E+007
.     .    1   index_scan           fr   PSX2 (s)              8.00E+007

--- SQL operation complete.
>>log;
```

CONTROL TABLE Statement

Considerations for CONTROL TABLE
Examples of CONTROL TABLE

The CONTROL TABLE statement specifies a performance-related option for DML accesses to a table or view. This statement can also be used as an embedded SQL compiler directive.

CONTROL TABLE is an SQL/MX extension.

```
CONTROL TABLE {table | *} control-table-option

control-table-option is:
     MDAM {'ENABLE'|'ON'|'OFF'}
   | PRIORITY 'priority-value'
   | IF_LOCKED {'RETURN'|'WAIT'}
   | TABLELOCK {'ENABLE'|'ON'|'OFF'}
   | TIMEOUT 'timeout-value'
   | SIMILARITY_CHECK {'ON'|'OFF'}
   | RESET
```

table

is the name of the table or view to which the control option applies. You must specify the name with the same qualification as the name that appears in subsequent references to which the control option applies. For example, if you specify a fully qualified table name, the fully qualified name must appear in subsequent references.

`*`

specifies that the control option applies to all tables subsequently referenced in the current process. A CONTROL TABLE *table* statement overrides the effect of a CONTROL TABLE `*` statement for the specified table or view.

`MDAM {'ENABLE'|'ON'|'OFF'}`

specifies whether to use MDAM for subsequently compiled DML statements that access the index. *table* refers to the index for which you wish to force MDAM. If you use the table name instead of the index name, NonStop SQL/MX uses the table clustering key and no other keys are forced for MDAM.

`ENABLE`

directs NonStop SQL/MX to determine whether to use MDAM for the specified index. The default is ENABLE.

`ON`

directs NonStop SQL/MX to use MDAM.

`OFF`

directs NonStop SQL/MX not to use MDAM.

`PRIORITY '`*`priority-value`*`'`

specifies the priority for subsequent file system requests to the Data Access Manager (DAM). DAM uses the priority to ensure efficient performance within a mixed workload environment. Short-duration OLTP-type requests should specify a higher priority than any concurrent long-duration query requests.

`'`*`priority-value`*`'`

specifies the priority, an integer value from 1 to 9. The default priority is 3. You can use the highest possible value (9), but using this value can interfere with SQL/MX system-level activity.

`IF_LOCKED {'RETURN'|'WAIT'}`

specifies the result if you attempt to access data with the READ COMMITTED or SERIALIZABLE access and the data is locked by another user.

`RETURN`

returns file system error 73.

`WAIT`

directs NonStop SQL/MX to wait for the other user to release the lock, until the timeout period elapses. The default is WAIT.

`TABLELOCK {'ENABLE'|'ON'|'OFF'}`

specifies whether to use table locks for subsequently compiled DML statements that access the table or view.

`ENABLE`

directs NonStop SQL/MX to determine whether to use table locks for the specified table or view. The default is ENABLE.

`ON`

directs NonStop SQL/MX to use table locks.

`OFF`

directs NonStop SQL/MX to not use table locks.

`TIMEOUT '`*`timeout-value`*`'`

specifies the time in hundredths of seconds allowed to complete file system requests from DML operations. If the time elapses before the file system can grant a request to lock data, the DML statement fails, and NonStop SQL/MX returns an error. This option is for static operations only. For dynamic operations, see SET TABLE TIMEOUT Statement on page 2-372.

`'timeout-value'`

specifies the time in hundredths of seconds. The range of values is from -1 to 2147483519, expressed in hundredths of seconds. The value -1 directs NonStop SQL/MX not to time out. The value 0 directs NonStop SQL/MX not to wait for a table lock. If the lock cannot be acquired, NonStop SQL/MX immediately returns an error.

`SIMILARITY_CHECK {'ON'|'OFF'}`

specifies whether to perform similarity checks for a new and previous table to avoid statement recompilation or always recompile at run time, depending on the outcome of various factors. This option applies only to tables (not views).

`ON`

directs NonStop SQL/MX to perform similarity checks at run time to determine whether the new table is similar to the previous table. If the tables are similar, NonStop SQL/MX uses the new table without statement recompilation. Otherwise, the SQL statement is recompiled with the new table name. The default is ON.

`OFF`

directs NonStop SQL/MX to recompile an SQL statement at run time, depending on the outcome of late name resolution, timestamp comparison, or table redefinition.

In MXCI, you can change a table in a prepared statement by using DEFINE commands. See ADD DEFINE Command on page 4-4 and ALTER DEFINE Command on page 4-6.

For more information about similarity checks, see the *SQL/MX Programming Manual for C and COBOL*.

`RESET`

cancels all previously set control options for the specified table or view, and NonStop SQL/MX uses only the default control values.

Considerations for CONTROL TABLE

Scope of CONTROL TABLE

The result of the execution of a CONTROL TABLE statement stays in effect until the current process terminates or until the execution of another CONTROL TABLE statement for the same *control-table-option* overrides it. For a detailed list of the precedence of default settings, see System Defaults Table on page 10-37.

A CONTROL TABLE *table* statement overrides the effect of a CONTROL TABLE * statement for the specified table or view.

Relationship to CONTROL QUERY DEFAULT

Use the CONTROL TABLE statement to override system-level default settings for the TABLELOCK, TIMEOUT, and SIMILARITY_CHECK attributes per table or for all tables in the current process. Use the CONTROL QUERY DEFAULT statement to override system-level default settings for these attributes and other attributes for all tables in the current process. For example, the statement `CONTROL TABLE * TIMEOUT '3000'` has the same effect as the statement `CONTROL QUERY DEFAULT TIMEOUT '3000'`. See CONTROL QUERY DEFAULT Statement on page 2-60.

The CONTROL TABLE statement has precedence over the CONTROL QUERY DEFAULT statement. For a detailed list of the precedence of default settings, see System Defaults Table on page 10-37.

Examples of CONTROL TABLE

- Turn off MDAM for the JOB table:

  ```
  CONTROL TABLE PERSNL.JOB MDAM 'OFF';
  ```

  If you want to enable or turn on MDAM for subsequent queries, you must specify the table name in the same way; for example, PERSNL.JOB.

- Set the length of the timeout for the current process to 30 seconds. To do so, change the static TIMEOUT attribute for the current process for all referenced tables and views. This setting overrides any system-level default settings for the TIMEOUT attribute.

  ```
  CONTROL TABLE * TIMEOUT '3000';
  ```

  The value 3000 is in hundredths of seconds, which is equivalent to 30 seconds.

- Cancel all previously set control options and use only the system-defined default setting:

  ```
  CONTROL TABLE * RESET;
  ```

CREATE CATALOG Statement

Considerations for CREATE CATALOG
Examples of CREATE CATALOG

The CREATE CATALOG statement creates a new SQL/MX catalog. See Catalogs on page 6-3.

CREATE CATALOG is an SQL/MX extension.

```
CREATE CATALOG catalog [LOCATION [\node.]$volume]
```

Syntax Description of CREATE CATALOG

catalog

is an SQL identifier that specifies the name of the new catalog. *catalog* must be unique among catalog names on the node.

LOCATION [*node.*]*$volume*

specifies the location of the metadata tables for the catalog.

node

is the name of the local node.

volume

is the name of an audited, non-SMF DAM volume on the specified node (or the Guardian volume named in the =_DEFAULTS define if no volume is specified).

If you do not specify a LOCATION clause and your system does not have a value for the DDL_DEFAULT_LOCATIONS default (either in your environment or at the system level) and your environment does not have a =_DEFAULTS define value, the CREATE statement will fail with an error.

Considerations for CREATE CATALOG

Reserved Catalogs

Catalog names beginning with NONSTOP_SQLMX_ are reserved for system metadata. You are not allowed to create (or to drop) catalogs with these reserved names.

Authorization and Availability Requirements

The users who are granted the privilege using GRANT CREATE CATALOG can create a catalog. If such users are not specified, any user can execute this statement.

Examples of CREATE CATALOG

- This example creates a catalog:

  ```
  CREATE CATALOG samdbcat;
  ```

CREATE INDEX Statement

Considerations for CREATE INDEX
Examples of CREATE INDEX

The CREATE INDEX statement creates an SQL/MX index based on one or more columns of a table. See Database Object Names on page 6-13.

CREATE INDEX is an SQL/MX extension.

```
CREATE [UNIQUE] INDEX index ON table
   (column-name [ASC[ENDING] | DESC[ENDING]]
      [,column-name [ASC[ENDING] | DESC[ENDING]]]...)
   [populate-option]
   [file-option]...

populate-option is: POPULATE | NO POPULATE

file-option is:
     LOCATION [\node.]$volume[.subvolume.file-name]
       [NAME partition-name]
   | partn-file-option
   | ATTRIBUTE[S] attribute [,attribute]...

partn-file-option is:
   {[RANGE] PARTITION
    [BY (partitioning-column  [,partitioning-column]...)]
     [(ADD range-partn-defn [,ADD range-partn-defn]...)]

   | HASH PARTITION
     [BY (partitioning-column [,partitioning-column]...)]
     [(ADD partn-defn [,ADD partn-defn]...)]}

range-partn-defn is:
  FIRST KEY {col-value | (col-value [,col-value]...)}
    partn-defn

partn-defn is:
   LOCATION [\node.]$volume[.subvolume.file-name]
     [EXTENT ext-size | (pri-ext-size [,sec-ext-size])]
     [MAXEXTENTS num-extents]
     [NAME  partition-name]

attribute is:
     ALLOCATE num-extents
   | {AUDITCOMPRESS | NO AUDITCOMPRESS}
   | BLOCKSIZE number-bytes
   | {CLEARONPURGE | NO CLEARONPURGE}
   | EXTENT ext-size | (pri-ext-size [,sec-ext-size])
   | MAXEXTENTS num-extents
```

Syntax Description of CREATE INDEX

`UNIQUE`

specifies that the values (including NULL values) in the column or set of columns that make up the index field cannot contain more than one occurrence of the same value or set of values. For indexes with multiple columns, the values of the columns as a group determines uniqueness, not the values of the individual columns. If you omit UNIQUE, duplicate values are allowed. The columns you specify for the index need not be declared NOT NULL (note that this is unlike CREATE TABLE and ALTER TABLE, which do require all columns of a specified unique constraint to be NOT NULL).

`index`

is an SQL identifier that specifies the simple name for the new index. You cannot qualify *`index`* with its catalog and schema names. Indexes have their own namespace within a schema, so an index name might be the same as a table or constraint name. However, no two indexes in a schema can have the same name.

`table`

is the name of the table for which to create the index. See Database Object Names on page 6-13.

```
column-name [ASC[ENDING] | DESC[ENDING]]
   [,column-name [ASC[ENDING] | DESC[ENDING]]]...
```

specifies the columns in *`table`* to include in the index. The order of the columns in the index need not correspond to the order of the columns in the table.

ASCENDING or DESCENDING specifies the storage and retrieval order for rows in the index. The default is ASCENDING.

Rows are ordered by values in the first column specified for the index. If multiple index rows share the same value for the first column, the values in the second column are used to order the rows, and so forth. If duplicate index rows occur in a nonunique index, their order is based on the sequence specified for the columns of the key of the underlying table. For ordering (but not for other purposes), nulls are greater than other values.

`populate-option`

`NO POPULATE`

specifies that the index is not to be populated when it is created. The index's partition(s) are created, but no data is written to the index, and it is marked “offline”. You can drop an offline index with the DROP INDEX statement. The DROP TABLE statement also drops offline indexes of the specified table. DML statements have no effect on offline indexes. If an index is created with the intention of using it for a constraint, it must be populated before creating the constraint. You can populate an offline index and remove its offline designation by using the POPULATE INDEX utility.

`POPULATE`

Specifies that the index is to be created and populated. If you omit the *populate-option*, the default is POPULATE

```
LOCATION [\node.]$volume[.subvolume.file-name]
  [NAME partition-name]
```

specifies a node and volume for the primary partition of the index.

node

is the name of a node on the Expand network.

For Guardian files representing a table or index partition or a view label, *node* can be any node from which the object's catalog is visible.

volume

is the name of an audited, non-SMF DAM volume on the specified node (or the Guardian volume named in the =_DEFAULTS define if no volume is specified).

If you do not specify a LOCATION clause and your system does not have a value for the DDL_DEFAULT_LOCATIONS default (either in your environment or at the system level) and your environment does not have a =_DEFAULTS value, the CREATE statement will fail with an error.

subvolume

is the designated schema subvolume for the schema in which the index is being created.

Follow these guidelines when using SQL/MX subvolume names:

- The name must begin with the letters *ZSD*, followed by a letter, not a digit (for example, ZSDa, not ZSD2).
- The name must be exactly eight characters.

file-name

is an optional Guardian file name. *file-name* names must be 8 characters in length and must end with the digits “00” (zero zero).

Any Guardian file name you specify must match the designated schema subvolume name for the schema in which the object is being created. Otherwise, NonStop SQL/MX returns an error.

partition-name

is an SQL identifier for a partition.

partn-file-option is:

```
{[RANGE] PARTITION
   [BY (partitioning-column  [,partitioning-column]...)]
   [(ADD range-partn-defn [,ADD range-partn-defn]...)]
```

defines secondary partitions for a range partitioned table. See Partitions on page 6-83 for details about partitions.

`BY (`*partitioning-column* `[,`*partitioning-column*`]...)`

specifies the partitioning columns. The default is the default partitioning key created by the STORE BY clause. Partitioning character columns must derive from the ISO88591 character set. Partitioning columns cannot be floating-point data columns.

```
| HASH PARTITION
   [BY (partitioning-column [,partitioning-column]...)]
   [(ADD partn-defn [,ADD partn-defn]...)]}
```

defines secondary partitions for a hash partitioned table. See Partitions on page 6-83 for details about partitions.

`BY (`*partitioning-column* `[,`*partitioning-column*`]...)`

specifies the columns that make up the partitioning key. If you do not specify this clause, the partitioning key is the same as the clustering key of the table. Partitioning columns cannot be floating-point data columns.

`ADD` *range-partn-defn*

defines a single secondary partition and includes the FIRST KEY and a *partn-defn*.

range-partn-defn is:

FIRST KEY {*col-value* | (*col-value* [,*col-value*]...)} *partn-defn*

specifies the beginning of the range for a range partitioned table. The FIRST KEY clause specifies the lowest values in the partition for columns stored in ascending order and the highest values in the partition for columns stored in descending order. These column values are referred to as the *partitioning key*.

col-value is a literal that specifies the first value allowed in the associated partition for that column of the partitioning key. If there are more storage key columns than *col-value* items, the first key value for each remaining key column is the lowest or highest value for the data type of the column (the lowest value for an ascending column and the highest value for a descending column). *col-value* must contain characters only from the ISO88591 character set.

If the index has a system-generated SYSKEY, its column list cannot consist only of column SYSKEY. The SYSKEY must be the last column of the column list, and you cannot specify a FIRST KEY value for the SYSKEY column. This limitation does not apply to a user-created SYSKEY column.

`ADD` *`partn-defn`*

defines a single secondary hash partition and includes the LOCATION of the partition.

`partn-defn` is:

```
LOCATION [\node.]$volume[.subvolume.file-name]
[EXTENT ext-size | (pri-ext-size [,sec-ext-size])]
[MAXEXTENTS num-extents]
    [NAME partition-name]
```

specifies a volume and optionally the node, subvolume, and filename for the partition.

`node`

is the name of a node on the Expand network. For Guardian files representing a table or index partition, a view label, or a stored procedure *`node`* can be any node from which the object's catalog is visible.

`volume`

is the name of an audited, non-SMF DAM volume on the specified node (or the Guardian volume named in the =_DEFAULTS define if none is specified). If you do not specify a LOCATION clause, NonStop SQL/MX uses the default volume named in the =_DEFAULTS define.

If you do not specify a LOCATION clause and your system does not have a value for the DDL_DEFAULT_LOCATIONS default (either in your environment or at the system level) and your environment does not have a =_DEFAULTS value, the CREATE statement will fail with an error.

You can locate more than one partition of an index on a single disk volume.

`subvolume`

is the designated schema subvolume for the schema in which the index is being created. Follow these guidelines when using SQL/MX subvolume names:

- The name must begin with the letters *`ZSD`*, followed by a letter, not a digit (for example, ZSDa, not ZSD2).
- The name must be exactly eight characters.

file-name

is an optional Guardian file name. *file-name* names must be 8 characters in length and must end with the digits “00” (zero zero).

Any Guardian file name you specify must match the designated schema subvolume name for the schema in which the object is being created. Otherwise, NonStop SQL/MX returns an error.

partition-name

is an SQL identifier for a partition.

partn-file-option is an SQL/MX extension.

See PARTITION Clause on page 7-6.

`ATTRIBUTE[S]` *attribute* `[,`*attribute*`]...`

specifies file attributes for the key-sequenced file that holds the index. In an ATTRIBUTES clause that is within a PARTITION clause, you must separate *attributes* with a space. In ATTRIBUTES clauses in other places, you can separate *attributes* with either a space or a comma. File attributes you can specify are:

ALLOCATE/DEALLOCATE on page 9-2	Controls amount of disk space allocated.
AUDITCOMPRESS on page 9-3	Controls whether unchanged columns are included in audit records.
BLOCKSIZE on page 9-4	Sets size of data blocks.
CLEARONPURGE on page 9-5 *	Controls disk erasure when index is dropped.
EXTENT on page 9-6	Controls size of extents that are allocated on disk.
MAXEXTENTS on page 9-7	Controls key compression in DP2 index blocks.
MAXEXTENTS on page 9-7	Controls maximum disk space to be allocated.

Attributes marked with an asterisk (*) default to the same value as the corresponding attribute in the underlying table. For more detail, see the entry for a specific attribute.

Considerations for CREATE INDEX

If you are creating an index on a large SQL/MX table that is already populated, you should use the NO POPULATE option, and then run the POPULATE INDEX utility to load the index. Because CREATE INDEX executes in a single TMF transaction, it could experience TMF limitations such as a transaction timeout if a large amount of data is to be moved. For more information about creating and populating indexes, see the *SQL/MX Installation and Management Guide*.

After you perform POPULATE INDEX, you should perform a FUP RELOAD on the index and all its partitions, to organize the index structure more efficiently and to reduce index levels.

If the MAXEXTENT value that you specified is too small, when you run the POPULATE INDEX utility it automatically increases the value to the largest possible size. When POPULATE INDEX completes it adjusts the MAXEXTENTS value to the value you specified, if it is greater than the number of extents that needed to be allocated. If the number of extents that needed to be allocated is greater than the value you specified, POPULATE INDEX adjusts the value for MAXEXTENTS to a value equal to the number of extents that it allocated, plus 50. This is similar to NonStop SQL/MP's behavior.

Authorization and Availability Requirements

To create an SQL/MX index, you must own the schema for the underlying table or be the super ID or object owner for the underlying table, and have access to all partitions of the underlying table.

CREATE INDEX locks out INSERT, DELETE, and UPDATE operations on the table being indexed. If other processes have rows in the table locked when the operation begins, CREATE INDEX waits until its lock request is granted or timeout occurs.

An index always has the same security as the table it indexes, so users authorized to access the table can also access the index. You cannot access an index directly.

Limits on Indexes

For nonunique indexes, the sum of the lengths of the columns in the index plus the sum of the length of the clustering key of the underlying table cannot exceed 2010 bytes for 4K blocks and 2048 bytes for 32K blocks. For unique indexes, the sum of the lengths of the columns in the index cannot exceed 2010 bytes for 4K blocks and 2048 bytes for 32K blocks.

There is no restriction on the number of indexes per table.

There is no restriction on the number of partitions an index supports.

Examples of CREATE INDEX

- This example creates an index on two columns of a table:

```
CREATE INDEX xempname
  ON persnl.employee (last_name, first_name);
```

- This example creates and partitions a unique index (one that could be used to support a UNIQUE, PRIMARY KEY, or referential constraint) on a table:

```
CREATE UNIQUE INDEX XEMP
ON PERSNL.EMPLOYEE (LAST_NAME, EMPNUM)
LOCATION $data1
ATTRIBUTE NO AUDITCOMPRESS
PARTITION (ADD FIRST KEY 'E' LOCATION $data1,
           ADD FIRST KEY 'J' LOCATION $data2,
           ADD FIRST KEY 'O' LOCATION $data2,
           ADD FIRST KEY 'T' LOCATION $data3);
```

CREATE PROCEDURE Statement

Considerations for CREATE PROCEDURE
Examples of CREATE PROCEDURE

The CREATE PROCEDURE statement registers an existing Java method as a stored procedure in Java (SPJ) within NonStop SQL/MX. To develop, deploy, and manage SPJs in SQL/MX, see the *SQL/MX Guide to Stored Procedures in Java*.

```
CREATE PROCEDURE procedure-ref([sql-parameter-list])
   EXTERNAL NAME 'java-method-name [([java-signature])]'
   EXTERNAL PATH 'class-or-JAR-file-path'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   [LOCATION procedure-label]
   [CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA
    | NO SQL]
   [DYNAMIC RESULT SETS max-result-sets]
   [NOT DETERMINISTIC | DETERMINISTIC]
   [ISOLATE | NO ISOLATE]

procedure-ref is:
   [[catalog-name.]schema-name.]procedure-name

sql-parameter-list is:
   sql-parameter[{, sql-parameter}...]

sql-parameter is:
   [parameter-mode] [sql-identifier] sql-datatype

parameter-mode is:
   IN
 | OUT
 | INOUT

java-method-name is:
   [package-name.]class-name.method-name

java-signature is:
   java-datatype[{, java-datatype}...]

procedure-label is:
   [\node.]$volume[.subvolume.filename]
```

Note. Delimited variables in this syntax diagram are case-sensitive. Case-sensitive variables include *java-method-name*, *java-signature*, *class-or-JAR-file-path*, and delimited parts of the *procedure-name*. The remaining syntax is not case-sensitive.

The *max-result-set* can have a value in the range 0–255 from J06.05 and later J-series RVUs and H06.16 and later H-series RVUs.

`procedure-ref([sql-parameter[{, sql-parameter}...]])`

specifies the name of the SPJ and SQL parameters that correspond to the signature of the SPJ method.

`procedure-ref`

specifies an ANSI logical name of the form:

`[[catalog-name.]schema-name.]procedure-name`

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

The *`procedure-name`* must be unique among the names of tables, views, SQL/MP aliases, sequence generators, and procedures within its schema. NonStop SQL/MX does not support the overloading of procedure names. That is, you cannot register the same procedure name more than once with different underlying SPJ methods.

You cannot prefix the procedure name with the name of a user metadata (UMD) table. For example, you cannot create a procedure named HISTOGRAMS_MYPROC. These names are reserved for user metadata.

If you do not fully qualify the procedure name, NonStop SQL/MX qualifies it according to the current settings of CATALOG and SCHEMA. If you set the NAMETYPE attribute to NSK instead of ANSI and you do not fully qualify the procedure name, NonStop SQL/MX returns an error. For more information on the CATALOG, SCHEMA, and NAMETYPE attributes, see the System Defaults Table on page 10-37.

`sql-parameter`

specifies an SQL parameter that corresponds to the signature of the SPJ method:

`[parameter-mode] [sql-identifier] sql-datatype`

`parameter-mode`

specifies the mode `IN`, `OUT`, or `INOUT` of a parameter. The default is `IN`.

`IN`

specifies a parameter that passes data to an SPJ.

`INOUT`

specifies a parameter that passes data to and accepts data from an SPJ.

`OUT`

specifies a parameter that accepts data from an SPJ.

sql-identifier

specifies an SQL identifier that describes the parameter. For more information, see Identifiers on page 6-56.

sql-datatype

specifies an SQL data type that corresponds to the Java parameter of the SPJ method. *sql-datatype* can be:

SQL/MX Data Type	Maps to Java Data Type...
CHAR[ACTER]* CHAR[ACTER] VARYING * VARCHAR* Ext PIC[TURE] X * NCHAR NCHAR VARYING NATIONAL CHAR[ACTER] NATIONAL CHAR[ACTER] VARYING	java.lang.String
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
NUMERIC ** DEC[IMAL]** Ext PIC[TURE] S9	java.math.BigDecimal
SMALLINT**	short
INT[EGER]**	INT (or java.lang.Integer if specified)***
LARGEINT	long (or java.lang.Long if specified)***
FLOAT	double (or java.lang.Double if specified)***
REAL	float (or java.lang.Float if specified)***
DOUBLE PRECISION	double (or java.lang.Double if specified)***

* The character set for character string data types can be ISO88591 or UCS2.
** Numeric data types can be only SIGNED, which is the default in NonStop SQL/MX.
*** By default, the SQL/MX data type maps to a Java primitive data type. The SQL/MX data type maps to a Java wrapper class only if you specify the wrapper class in the Java signature of the EXTERNAL NAME clause.

Ext	This icon indicates an SQL data type that is an SQL/MX extension to the ANSI standard. All other SQL data types in this table conform to the ANSI standard.

For more information, see Data Types on page 6-17.

`EXTERNAL NAME` '*java-method-name* [*java-signature*]'

java-method-name

specifies the case-sensitive name of the SPJ method of the form:

[*package-name*.]*class-name*.*method-name*

The Java method must exist in a Java class file, *class-name.class*, in the OSS directory, or the JAR file path specified by the EXTERNAL PATH clause. The Java method must be defined as `public` and `static` and have a return type of `void`. For guidelines on how to write an SPJ method, see the *SQL/MX Guide to Stored Procedures in Java*.

If the class file that contains the SPJ method is part of a package, you must also specify the package name. If you do not specify the package name, the CREATE PROCEDURE statement fails to register the SPJ.

java-signature

specifies the signature of the SPJ method and consists of:

([*java-datatype*[{, *java-datatype*}...]])

The Java signature is necessary only if you want to specify a Java wrapper class (for example, `java.lang.Integer`) instead of a Java primitive data type (for example, `INT`). An SQL/MX data type maps to a Java primitive data type by default.

The Java signature is case-sensitive and must be placed within parentheses, such as `(java.lang.Integer, java.lang.Integer)`. The signature must specify each of the parameter data types in the order they appear in the Java method definition within the class file. Each Java data type that corresponds to an OUT or INOUT parameter must be followed by empty square brackets ([ ]), such as `java.lang.Integer[]`.

java-datatype

specifies a mappable Java data type. For the mapping of the Java data types to SQL/MX data types, see sql-datatype on page 2-90.

`EXTERNAL PATH` '*class-file-path*'

specifies a case-sensitive string identifying the OSS directory or the JAR file path where the Java class file that contains the SPJ method resides.

Specify package names in the EXTERNAL NAME clause, not the EXTERNAL PATH clause.

`LANGUAGE JAVA`

specifies that the external user-defined routine is written in the Java language.

`PARAMETER STYLE JAVA`

specifies that the run-time conventions for arguments passed to the external user-defined routine are those of the Java language.

`LOCATION` *`procedure-label`*

specifies a Guardian physical name and location for the stored procedure label. For more information on the procedure label, see the *SQL/MX Guide to Stored Procedures in Java*.

`[\`*`node`*`.]$`*`volume`*`[.`*`subvolume`*`.`*`filename`*`]`

`node`

is the name of a node on the Expand network. The *node* can be any node where the catalog of the SPJ is visible.

`volume`

If you do not specify a LOCATION clause and your system does not have a value for the DDL_DEFAULT_LOCATIONS default (either in your environment or at the system level) and environment does not have a =_DEFAULTS value, the CREATE statement fails with an error.

`subvolume`

is the designated schema subvolume for the schema in which the SPJ is being created. Follow these guidelines when using SQL/MX subvolume names:

- The name must begin with the letters *ZSD*, followed by a letter, not a digit.
- The name must be exactly eight characters long.

Any Guardian file name you specify must match the designated schema subvolume name for the schema in which the SPJ is being created. Otherwise, NonStop SQL/MX returns an error.

`filename`

is an optional Guardian file name. The name must be eight characters long in length and must end with the digits “00” (zero zero).

`NO SQL`

specifies that the SPJ cannot perform SQL operations.

`CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA`

specifies that the SPJ can perform SQL operations. Currently, all the options allow an SPJ to read and modify SQL data. If you do not specify an SQL access mode, the default is `CONTAINS SQL`.

`DYNAMIC RESULT SETS` *`max-result-sets`*

specifies the maximum number of result sets that the SPJ can return. If you specify this clause, you must set the value in the range 0 through 255.

`NOT DETERMINISTIC | DETERMINISTIC`

specifies whether the SPJ always returns the same values for OUT and INOUT parameters for a given set of argument values (`DETERMINISTIC`) or does not return the same values (`NOT DETERMINISTIC`, the default option). For a deterministic SPJ, the database server reserves the right to cache the results of a CALL statement and reuse them during subsequent calls, optimizing the CALL statement. NonStop SQL/MX allows both options but always treats the SPJ as nondeterministic.

`ISOLATE | NO ISOLATE`

specifies that the SPJ executes either in the environment of the database server (`NO ISOLATE`) or in an isolated environment (`ISOLATE`, the default option). NonStop SQL/MX allows both options but always executes the SPJ in the SQL/MX UDR server process (`ISOLATE`).

Considerations for CREATE PROCEDURE

Authorization and Availability Requirements

To issue a CREATE PROCEDURE statement, you must be the owner of the schema, or be the super ID, and have read access to the Java class file or JAR file that contains the SPJ method.

Examples of CREATE PROCEDURE

- This CREATE PROCEDURE statement registers the SPJ named LOWERPRICE, which does not accept any arguments:

```
SET CATALOG samdbcat;
SET SCHEMA sales;

CREATE PROCEDURE lowerprice()
   EXTERNAL NAME 'Sales.lowerPrice()'
   EXTERNAL PATH '/usr/mydir/myclasses'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA;
```

The statement verifies the existence of the SPJ method, `lowerprice`, in the `/usr/mydir/myclasses/Sales.class`.

Because the procedure name is not qualified by a catalog and schema, NonStop SQL/MX qualifies it according to the current settings of CATALOG and SCHEMA, which are SAMDBCAT and SALES in this case. To call this SPJ, use this statement:

```
CALL lowerprice();
```

- This CREATE PROCEDURE statement registers the SPJ named TOTALPRICE, which accepts three input parameters and returns a numeric value, the total price, to an INOUT parameter:

```
CREATE PROCEDURE samdbcat.sales.totalprice(IN NUMERIC (18),
                                           IN VARCHAR (10),
                                  INOUT price NUMERIC (18,2))
   EXTERNAL NAME 'pkg.subpkg.Sales.totalPrice'
   EXTERNAL PATH '/usr/mydir/myJar.jar'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA;
```

To call this SPJ in MXCI, use these statements:

```
SET PARAM ?p 10.00;

CALL samdbcat.sales.totalprice(23, 'standard', ?p);

PRICE
---------------------

               253.96
```

- This CREATE PROCEDURE statement registers the SPJ named MONTHLYORDERS, which accepts an integer value for the month and returns the number of orders:

```
CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT,
                                              OUT number INT)
   EXTERNAL NAME
'Sales.numMonthlyOrders (INT, java.lang.Integer[])'
   EXTERNAL PATH '/usr/mydir/myclasses'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA;
```

Because the OUT parameter is supposed to map to the Java wrapper class, `java.lang.Integer`, you must specify the Java signature in the EXTERNAL NAME clause. To call this SPJ, use this statement:

```
CALL samdbcat.sales.monthlyorders(3, ?);

NUMBER
-----------
```

4

- This CREATE PROCEDURE statement registers the SPJ named SALES.ORDER_SUMMARY and returns two result sets.

```
CREATE PROCEDURE SAMDBCAT.SALES.ORDER_SUMMARY
  (
    IN ON_OR_AFTER_DATE VARCHAR(20) CHARACTER SET ISO88591
  , OUT NUM_ORDERS LARGEINT
  )

  DYNAMIC RESULT SETS 2
  READS SQL DATA LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'SPJMethods.orderSummary'
  EXTERNAL PATH '/usr/mydir/myclasses';
```

- This CREATE PROCEDURE statement registers the SPJ named SALES.PART_DATA and returns four result sets.

```
CREATE PROCEDURE SAMDBCAT.SALES.PART_DATA
  (
    IN PARTNUM NUMERIC(4)
  , OUT PARTDESC CHAR(18)
  , OUT PRICE NUMERIC(8,2)
  , OUT QTY_AVAIL NUMERIC(5)
  )
  DYNAMIC RESULT SETS 4
  READS SQL DATA LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'SPJMethods.partData'
  EXTERNAL PATH '/usr/mydir/myclasses';
```

For more examples, see the *SQL/MX Guide to Stored Procedures in Java*.

CREATE SCHEMA Statement

Considerations for CREATE SCHEMA
Examples of CREATE SCHEMA

CREATE SCHEMA creates an SQL/MX schema.

CREATE SCHEMA with the optional location clause is an SQL/MX extension.

```
CREATE SCHEMA schema-clause [schema-element
     [, schema-element] ...]]

schema-clause is:
     schema
   | schema AUTHORIZATION auth-id
   | schema AUTHORIZATION auth-id location-clause
   | schema  location-clause

location-clause is:
   LOCATION subvolume [reuse-clause]

reuse-clause is:
   REPEAT USE ALLOWED

schema-element is:
     table-definition
   | view-definition
   | grant-statement
   | index-definition
```

Syntax Description of CREATE SCHEMA

schema

is a schema name for the new schema. A simple schema name is an SQL identifier. A schema name can also be of the form *catalog-name.schema-name*.

AUTHORIZATION *auth-id*

specifies the owner of the schema. The default is the current authorization ID. The *auth-id* must be the current authorization ID unless the current authorization ID is a SUPER user. A SUPER user can specify any currently valid authorization ID as the owner of the schema.

Enter an *auth-id* as a simple name, for example, `"sql.user1".` Because of the "." , you must enclose the user name in double quotes, the same as for a delimited identifier.

You cannot specify PUBLIC as the *auth-id*, because a schema can have only one owner.

`LOCATION` *`subvolume`*

optionally specifies the designated subvolume name for the schema. Ordinarily, NonStop SQL/MX generates a subvolume name for the schema. However, in some instances, you might need to specify the subvolume, such as when creating an RDF backup database. In this case, the subvolume name for each backup schema must match the subvolume name for the corresponding schema in the primary database, and you must use the REPEAT USE ALLOWED clause for the statement to succeed.

In either case, the schema subvolume is written to the SCHEMATA.SCHEMA_SUBVOLUME column in the SQL/MX system schema. The schema subvolume is used as the subvolume for the locations of all objects created within that schema.

If the optional subvolume name is omitted, NonStop SQL/MX generates a subvolume name for the schema.

Follow these guidelines when using SQL/MX subvolume names:

- The name must begin with the letters *ZSD*, followed by a letter, not a digit (for example, ZSDa, not ZSD2).
- The name must be exactly eight characters long.
- All Guardian files representing data in a particular schema must have the same subvolume name regardless of the volume on which they reside. This subvolume name must match the subvolume name indicated in the system schema column SCHEMATA.SCHEMA_SUBVOLUME.
- For RDF database creation, if you explicitly specify the subvolume that is already in use by the primary database, use the REPEAT USE ALLOWED clause to avoid receiving an error when executing the statement.

Valid SQL/MX subvolume names are:

```
ZSDBMM3K
ZSDADMM8
```

`REPEAT USE ALLOWED`

indicates that NonStop SQL/MX should allow subvolume names to be reused. If the subvolume name is in use, the schema will be created anyway and you will receive a warning.

If you omit this clause, the subvolume name you enter must not be in use by any other schema. If the subvolume name has been used for another schema, you will receive an error.

`schema-element`

specifies the objects to be defined in the schema being created. The element in the statement must be in the same schema as the schema you are creating. Schema elements must appear in sequence—a schema element that depends on another schema element must be listed after that schema element.

table-definition

is a CREATE TABLE statement.

view-definition

is a CREATE VIEW statement.

grant-statement

is a GRANT statement.

index-definition

is a CREATE INDEX statement. *index-definition* cannot be the only *schema-element*.

Considerations for CREATE SCHEMA

Duplicate Schema Subvolume

One use of the schema subvolume is to identify all Guardian files in a schema for use with Guardian-based commands for TMF, RDF or other subsystems. Unless you use the REPEAT USE ALLOWED clause, NonStop SQL/MX prevents you from specifying a subvolume name that is already in use by another schema. If you use this clause you will receive a warning and the operation succeeds. This action is recommended only when creating an RDF backup database.

If you do reuse a schema subvolume, a Guardian wild card of the form `\system.$*.subvolume.*` will specify physical files from all schemas using this subvolume name. This might affect your future ability to refer only to objects in specific schemas when issuing commands to TMF or RDF.

Reserved Schema Names

Schema names that begin with DEFINITION_SCHEMA_VERSION_ are reserved (in all catalogs) for system metadata. You cannot create schemas with these names in user catalogs.

These names are not reserved (you can create schemas with these names in user catalogs): SYSTEM_SCHEMA, SYSTEM_DEFAULTS_SCHEMA, MXCS_SCHEMA, SYSTEM_SECURITY_SCHEMA, SYSTEM_SQLJ_SCHEMA.

Schemas named SYSTEM_SCHEMA, SYSTEM_DEFAULTS_SCHEMA, MXCS_SCHEMA, SYSTEM_SECURITY_SCHEMA, and SYSTEM_SQLJ_SCHEMA in the system catalog are reserved for metadata. You cannot drop them or create objects in them.

Authorization and Availability Requirements

A catalog owner and the users who are granted the privilege by using GRANT CREATE SCHEMA can create a schema. If such users are not specified, any user can execute this statement.

Examples of CREATE SCHEMA

- This example creates a schema:

```
CREATE SCHEMA mycat.myschema;
```

- This example creates a schema with pubs.jsmith as the owner, located on subvolume ZSDABCDE:

```
CREATE SCHEMA sch2 AUTHORIZATION “pubs.jsmith” LOCATION
ZSDABCDE;
```

- This example creates a schema located on subvolume ZSDSCHEM:

```
CREATE SCHEMA myschema LOCATION ZSDSCHEM;
```

- This example intentionally creates a schema located on subvolume ZSDSCHE2 when that subvolume is already in use by another schema:

```
CREATE SCHEMA myschema LOCATION ZSDSCHE2 REPEAT USE ALLOWED;
```

CREATE SEQUENCE Statement

Considerations for CREATE SEQUENCE
Examples of CREATE SEQUENCE

The CREATE SEQUENCE statement creates a sequence generator in the specified schema. The CREATE SEQUENCE statement must specify a name, but specifying the attributes such as LOCATION, START WITH, MAXVALUE, MINVALUE, INCREMENT BY, and CYCLE is optional.

Note. Each attribute can be specified at most once in the CREATE SEQUENCE statement. A warning is returned if you specify the ORDER and CACHE attributes.

```
CREATE SEQUENCE sequence [sequence-generator-options
     [sequence-generator-options] ...] [sequence-file-option]

sequence-generator-options are:
{ sequence-generator-data-type-option }
   | { sequence-generator-start-with-option }
   | { sequence-generator-increment-by-option }
   | { sequence-generator-maxvalue-option }
   | { sequence-generator-minvalue-option }
   | { sequence-generator-cycle-option }
   | { sequence-generator-cache-option }
   | { sequence-generator-order-option }

sequence-generator-data-type-option is:
{ NUMERIC [ (sg-precision [, 0]) ]
  [ SIGNED | UNSIGNED ]
  | SMALLINT [ SIGNED | UNSIGNED ]
  | INT[EGER] [ SIGNED | UNSIGNED ]
  | LARGEINT [ SIGNED ]
}

sg-precision is: { 1,2,3,...,28 }

sequence-generator-start-with-option is:
{ START WITH sequence-generator-numeric-value }

sequence-generator-increment-by-option is:
{ INCREMENT BY sequence-generator-numeric-value }

sequence-generator-maxvalue-option is:
{ MAXVALUE sequence-generator-numeric-value
  | NOMAXVALUE
  | NO MAXVALUE
}

sequence-generator-minvalue-option is:
{ MINVALUE sequence-generator-numeric-value
  | NOMINVALUE
  | NO MINVALUE
}
```

```
sequence-generator-cycle-option is:
{ CYCLE  | NOCYCLE | NO CYCLE}

sequence-generator-cache-option is:
{ CACHE sequence-generator-numeric-value  | NOCACHE }

sequence-generator-order-option is: { ORDER  | NOORDER }

sequence-generator-numeric-value is: < numeric-literal >

sequence-file-option is:
LOCATION [\node.]$volume[.subvolume.file-name]
```

Syntax Description of CREATE SEQUENCE

sequence

specifies the ANSI name of the sequence generator.

sequence-generator-data-type-option

specifies the sequence generator data type. Specifying the data type is optional. The supported data types are:

- Signed or Unsigned SMALLINT
- Signed or Unsigned INTEGER
- Signed LARGEINT, which is the default data type
- Signed or Unsigned NUMERIC with maximum precision of 28

sequence-generator-start-with-option

specifies the first sequence number.

By default, the minimum value of the sequence generator data type is considered as the start value for an ascending sequence and the maximum value is considered as the start value for a descending sequence.

The start value must be less than or equal to MAXVALUE, and greater than or equal to MINVALUE.

sequence-generator-increment-by-option

specifies the increment value to be added to the CURRENT_VALUE to obtain the next value in the sequence. Default is 1 (one). This value cannot be 0 (zero). Sequences can be either ascending or descending. An ascending sequence is associated with a positive increment and a descending sequence is associated with a negative increment.

sequence-generator-maxvalue-option

specifies the MAXVALUE for the sequence generator. The default is the maximum value of the sequence generator data type.

If NO MAXVALUE or NOMAXVALUE is specified, MAXVALUE is the maximum value of the sequence generator data type.

This value must be greater than MINVALUE and must be greater than or equal to START WITH value.

sequence-generator-minvalue-option

specifies the MINVALUE for the sequence generator. The default is the minimum value of the sequence generator data type.

If NO MINVALUE or NOMINVALUE is specified, MINVALUE is the minimum value of the sequence generator data type.

This value must be less than MAXVALUE and must be less than or equal to START WITH value.

sequence-generator-cycle-option

specifies the behavior of the sequence generator after reaching either the maximum or minimum value. The default is NO CYCLE.

If NO CYCLE is specified, an exception is raised if a new value is requested from the sequence generator after:

- an ascending sequence reaches the maximum value
- a descending sequence reaches the minimum value

If CYCLE is specified, the sequence generator loops over to:

- generate the minimum value after an ascending sequence reaches the maximum value
- generate the maximum value after a descending sequence reaches the minimum value

sequence-generator-cache-option

supported syntactically.

sequence-generator-order-option

supported syntactically.

sequence-file-option

specifies the location.

Considerations for CREATE SEQUENCE

Authorization Requirements

To create a sequence generator, you must own its schema or be the Super ID.

Restrictions

- The CACHE and ORDER attributes are only supported syntactically. A warning is returned if you use them in the CREATE SEQUENCE statement.
- You cannot specify the INCREMENT BY, MAXVALUE or NOMAXVALUE or NO MAXVALUE, MINVALUE or NOMINVALUE or NO MINVALUE, CYCLE or NO CYCLE or NOCYCLE attributes more than once in the statement.

Reserved Names

The user metadata table names are reserved. You cannot create a sequence generator with these reserved names. For example, you cannot create a sequence generator named HISTOGRAMS.

Examples of CREATE SEQUENCE

- This example creates a sequence generator with default attribute values:

```
CREATE SEQUENCE CAT.SCH.MYSEQ LARGEINT
```

- This example creates a sequence generator specifying values for MINVALUE, MAXVALUE and INCREMENT BY attributes:

```
CREATE SEQUENCE CAT.SCH.MYSEQ LARGEINT
START WITH 100
INCREMENT BY 1
MINVALUE 100
MAXVALUE 1000000 ;
```

- This example creates a sequence generator in the specified Guardian location:

```
CREATE SEQUENCE SEQ2 LOCATION $DATA04;
```

CREATE SQLMP ALIAS Statement

Considerations for CREATE SQLMP ALIAS
Examples of CREATE SQLMP ALIAS

The CREATE SQLMP ALIAS statement defines mappings from an ANSI name to the physical name of an SQL/MP table or view.

CREATE SQLMP ALIAS is an SQL/MX extension.

```
CREATE SQLMP ALIAS catalog.schema.object
   [\node.]$volume.subvol.filename
```

catalog.schema.object

is the alias name of an SQL/MP table or view. *catalog* and *schema* denote ANSI-defined catalog and schema, and *object* is a simple name for the table or view. If any of the three parts of the name is an SQL/MX reserved word, you must delimit it by enclosing it in double quotes. For example: `mycat."sql".myview`.

See Catalogs on page 6-3, and Identifiers on page 6-56.

[\node.]$volume.subvol.filename

is the fully qualified Guardian physical name of a table, view, or partition.

In this four-part name, *node* is the name of a node of a NonStop server, *$volume* is the name of a disk volume, *subvol* is the name of a subvolume, and *filename* is the name of an existing SQL/MP table or view. *node* is not required to be the local node. if any of the four parts of the name is an SQL/MX reserved word, you must delimit it by enclosing it in double quotes. Such delimited parts are not case-sensitive. For example: `$myvol."join".mytab`.

If you do not specify `\`*node*, the default is the Guardian node named in the =_DEFAULTS define. The value for the physical name is upshifted when the row is inserted into the SQLMX metadata table.

If the underlying file does not exist or is not an SQL/MP table or view, NonStop SQL/MX returns an error.

The *object* part of the name cannot have the name of a UMD table as a prefix. For example, it cannot be HISTOGRAMS_MYALIAS.

Considerations for CREATE SQLMP ALIAS

Reserved Alias Names

Alias names prefixed by the name of a UMD table are reserved. You cannot create aliases with such names. For example, you cannot create an alias named HISTOGRAMS_MYALIAS.

Usage Restrictions

If the catalog and schema do not exist, NonStop SQL/MX returns an error.

If the specified alias name already exists, NonStop SQL/MX returns an error.

You can map the same SQL/MP table or view to multiple different ANSI names in the same catalog and schema and in different catalogs and schemas. For example, you can create these four mappings for a single SQL/MP table or view:

```
cat1.sch1.obj1
cat1.sch1.obj2
cat2.sch1.obj2
cat2.sch2.obj2
```

In SQL/MX releases earlier than SQL/MX Release 2.x, if you re-mapped a table without dropping the SQLMP alias, NonStop SQL/MX would issue an error. In SQL/MX Releases 2.x you can map multiple ANSI names to one SQL/MP object, and access the same MP object using different alias names.

Only these DDL statements allow the use of an SQL/MP alias name: CREATE SQLMP ALIAS, DROP SQLMP ALIAS, ALTER SQLMP ALIAS, and UPDATE STATISTICS.

Moving and dropping of the underlying SQL/MP object does not result in altering or dropping the associated SQLMP aliases. See Considerations for DROP SQLMP ALIAS for details.

Managing Changes to SQLMP Aliases When SQL/MP Files Change

The alias information described in NonStop SQL/MX might become incorrect or orphaned. It becomes incorrect if the SQL/MP file name associated with the SQLMP ALIAS is moved to a different location. It becomes orphaned if the SQL/MP file name associated with the SQLMP ALIAS is removed. SQL/MP objects might be moved to different locations when users issue partition management commands with the SQL/MP ALTER statement or when they recover files to a different location by using TMF RECOVERY. SQL/MP files can be removed when users drop the table or view with an SQL/MP ALTER statement. You must alter the SQLMP ALIAS (if objects are moved) or drop the SQLMP ALIAS (if objects are dropped). See ALTER SEQUENCE Statement on page 2-13 and DROP SQLMP ALIAS Statement on page 2-188 for details.

Late Bind

If you compile an application that uses an SQL/MP alias and later you change the SQL/MP alias to map to a different SQL/MP table, the SQL/MP table definition is no longer compatible with the definition used at compile time. As a result, you must manually recompile applications that use the alias. If the late bind does not find the SQL/MP table underlying the SQL/MP alias or if the SQL/MP table was moved, NonStop SQL/MX returns an error.

For more information, see the *SQL/MX Programming Manual for C and COBOL*.

Embedding the Statement in an SQL Program

Embed

If you embed a CREATE SQLMP ALIAS statement in a static SQL program, subsequent references in the same program to the SQL/MP alias cause compilation errors because the alias does not reside in the OBJECTS table. To avoid these errors, create logical name mappings separately before compiling static SQL programs that refer to the SQL/MP alias. Compilation errors do not occur when you create and refer to SQL/MP aliases in a dynamic SQL program. For more information on embedding SQL in programs, see the *SQL/MX Programming Manual for C and COBOL*.

Partitioned Tables

Use the CREATE SQLMP ALIAS statement to create logical mappings for different partitions of a table. That is, two partitions of the same table can be referenced with different ANSI names. However, HP recommends that you map an alias to the primary partition for accessing the entire partitioned table.

Aliases can be visible to a remote node through a REGISTER CATALOG statement

Authorization and Availability Requirements

To create, alter, or drop aliases, you must be the owner of the schema or be the super ID.

Examples of CREATE SQLMP ALIAS

- Suppose that you have created an SQL/MP table by using this SQL/MP CREATE TABLE statement:

```
CREATE TABLE $myvol.mysubvol.mytable
    ( num    NUMERIC (4) UNSIGNED NOT NULL
     ,name    VARCHAR (20)
     ,PRIMARY KEY (num) );
```

  This statement creates a mapping in the metadata table:

```
CREATE SQLMP ALIAS mycatalog.myschema.mytable
                  $myvol.mysubvol.mytable;
```

CREATE TABLE Statement

Considerations for CREATE TABLE
Examples of CREATE TABLE

The CREATE TABLE statement creates an SQL/MX table. See Database Object Names on page 6-13.

```
   CREATE TABLE table
    { (table-element [,table-element]...) | like-spec }
    [file-option ]...

table-element is:
     column-definition
   | [CONSTRAINT constraint-name] table-constraint

column-definition is:
   column data-type
   [DEFAULT default | NO DEFAULT | identity-column-
specification]
   [HEADING 'heading-string' | NO HEADING]
   [[CONSTRAINT constraint-name] column-constraint]...

data-type is:
     CHAR[ACTER] [(length [CHARACTERS])]
         [CHARACTER SET char-set-name] [COLLATE DEFAULT]
         [UPSHIFT]
   | PIC[TURE] X [(length)] [CHARACTER SET char-set-name]
               [COLLATE DEFAULT] [DISPLAY] [UPSHIFT]
   | CHAR[ACTER] VARYING (length)
               [CHARACTER SET char-set-name]
               [COLLATE DEFAULT] [UPSHIFT]
   | VARCHAR (length) [CHARACTER SET char-set-name]
               [COLLATE DEFAULT] [UPSHIFT]
   | PIC[TURE] [S]{ 9(integer) [V[9(scale)]] | V9(scale) }
               [DISPLAY [SIGN IS LEADING] | COMP]
   | NCHAR [(length) [COLLATE DEFAULT] [UPSHIFT]
   | NCHAR VARYING(length) [COLLATE DEFAULT] [UPSHIFT]
   | NUMERIC [(precision [,scale])] [SIGNED|UNSIGNED]
   | SMALLINT [SIGNED|UNSIGNED]
   | INT[EGER] [SIGNED|UNSIGNED]
   | LARGEINT
   | DEC[IMAL] [(precision [,scale])] [SIGNED|UNSIGNED]
   | FLOAT [(precision)]
   | REAL
   | DOUBLE PRECISION
   | DATE
   | TIME [(time-precision)]
   | TIMESTAMP [(timestamp-precision)]
   | INTERVAL { start-field TO end-field | single-field }
```

```
default is:
     literal
   | NULL
   | CURRENT_DATE
   | CURRENT_TIME
   | CURRENT_TIMESTAMP
   | {CURRENT_USER | USER}

identity-column-specification is:
     identity-type [(internal-sequence-generator-options)]
[sg-location]

identity-type is:
     GENERATED BY DEFAULT AS IDENTITY
   | GENERATED ALWAYS AS IDENTITY

internal-sequence-generator-options is:
     internal-sequence-generator-option ...

internal-sequence-generator-option is:
     internal-sequence-generator-start-with-option
   | basic-internal-sequence-generator-option

internal-sequence-generator-start-with-option is:
     START WITH signed-numeric-literal

basic-internal-sequence-generator-option is:
     internal-sequence-generator-increment-by-option
   | internal-sequence-generator-maxvalue-option
   | internal-sequence-generator-minvalue-option
   | internal-sequence-generator-cycle-option

internal-sequence-generator-increment-by-option is:
     INCREMENT BY signed-numeric-literal

internal-sequence-generator-maxvalue-option is:
     MAXVALUE signed-numeric-literal
   | NO MAXVALUE

internal-sequence-generator-minvalue-option is:
     MINVALUE signed-numeric-literal
   | NO MINVALUE

internal-sequence-generator-cycle-option is:
     NO CYCLE

sg-location is:
     LOCATION [\node.]$volume[.subvolume.file-name]
```

```
column-constraint is:
     NOT NULL [[NOT] DROPPABLE]
   | UNIQUE
   | PRIMARY KEY [ASC[ENDING] | DESC[ENDING]]
        [[NOT] DROPPABLE]
   | CHECK (condition)
   | REFERENCES ref-spec
 ref-spec is:
   referenced-table [(column-list)]
  [referential triggered action]

referential triggered action is:
     update rule [delete rule]
   | delete rule [update rule]

update rule is: ON UPDATE referential action

delete rule is: ON DELETE referential action

referential action is:  RESTRICT
   | NO ACTION
   | CASCADE
   | SET NULL
   | SET DEFAULT

column-list is:
   column-name [,column-name]...

table-constraint is:
     UNIQUE (column-list)
   | PRIMARY KEY (key-column-list) [[NOT] DROPPABLE]
   | CHECK (condition)
   | FOREIGN KEY (column-list) REFERENCES ref-spec

key-column-list is:
   column-name [ASC[ENDING] | DESC[ENDING]]
      [,column-name [ASC[ENDING] | DESC[ENDING]]]...
```

```
file-option is:
     STORE BY store-option
   | LOCATION [\node.]$volume[.subvolume.file-name]
     [NAME partition-name]
   | partn-file-option
   | ATTRIBUTE[S] attribute [,attribute]...

store-option is:
     PRIMARY KEY
   | (key-column-list)

partn-file-option is:
    { [RANGE] PARTITION
      [BY (partitioning-column  [,partitioning-column]...)]
        [(ADD range-partn-defn [,ADD range-partn-defn]...)]

     | HASH PARTITION
       [BY (partitioning-column [,partitioning-column]...)]
        [(ADD partn-defn [,ADD partn-defn]...)]}

range-partn-defn is:
   FIRST KEY {col-value | (col-value [,col-value ]...)}
    partn-defn

partn-defn is:
   LOCATION [\node.]$volume[.subvolume.file-name]
     [EXTENT ext-size | (pri-ext-size [,sec-ext-size])]
     [MAXEXTENTS num-extents]
     [NAME partition-name]

attribute is:
      ALLOCATE num-extents
   | {AUDITCOMPRESS | NO AUDITCOMPRESS}
   |  BLOCKSIZE number-bytes
   | {CLEARONPURGE | NO CLEARONPURGE}
   |  EXTENT ext-size | (pri-ext-size [,sec-ext-size])
   |  MAXEXTENTS num-extents

like-spec is:
   LIKE source-table [include-option]...

include-option is:
     WITH CONSTRAINTS
   | WITH HEADINGS
   | WITH PARTITIONS

include-option is:
     WITH CONSTRAINTS | WITH PARTITIONS
```

Syntax Description of CREATE TABLE

table

is the ANSI logical name for the new table and must be unique among names of tables, views, SQL/MP aliases, and procedures within its schema. You cannot specify a Guardian physical name although you can specify it with the LOCATION clause.

column data-type

specifies the name and data type for a column in the table. At least one column definition is required in a CREATE TABLE statement.

column is an SQL identifier. *column* must be unique among column names in the table. If the name is an SQL/MX reserved word, you must delimit it by enclosing it in double quotes. Such delimited parts are case-sensitive. For example: `"join"`.

column cannot be SYSKEY if the clustering key contains SYSKEY.

data-type is the data type of the values that can be stored in *column*. A default value must be of the same type as the column, including the character set for a character column. See Data Types on page 6-17.

`DEFAULT` *default* `| NO DEFAULT`

specifies a default value for the column or specifies that the column does not have a default value. See DEFAULT Clause on page 7-2.

identity-column-specification

specifies that the column is an IDENTITY column indicating that the system can generate values for it using the internal Sequence Generator options. Nonstop SQL/MX will generate unique values for this column by default. See IDENTITY Column and internal Sequence Generators on page 2-129 and Generating Values for an IDENTITY Column on page 2-132.

```
GENERATED BY DEFAULT AS IDENTITY | GENERATED ALWAYS AS
IDENTITY
```

- GENERATED BY DEFAULT AS IDENTITY

  creates an IDENTITY column. This type accepts both system-generated values and user-supplied values for the IDENTITY column. The system generates a value for the IDENTITY column if the user does not provide one. The system does not guarantee unique values for the IDENTITY column if the user supplies a value for the IDENTITY column. The internal Sequence Generator is created when an IDENTITY column is specified.

- GENERATED ALWAYS AS IDENTITY

  creates an IDENTITY column. This type accepts only system-generated values. A user-supplied value is not allowed. The system guarantees unique values for the IDENTITY column. The internal Sequence Generator is created when an IDENTITY column is specified.

`START WITH` *`signed-numeric-literal`*

start value of the cycle range for the IDENTITY column. The default is the value provided in MINVALUE option of the internal Sequence Generator. If the MINVALUE is omitted or NO MINVALUE is specified, the default is 0 (zero).

`INCREMENT BY` *`signed-numeric-literal`*

increments the `CURRENT_VALUE` by *`signed-numeric-literal`* to obtain the next value in the sequence. See SG Table on page 2-130. Default is 1 (one). The INCREMENT BY value cannot be greater than the maximum value of the data type of the IDENTITY column.

`MAXVALUE` *`signed-numeric-literal`* `| NO MAXVALUE`

the non-negative value of the data type of the IDENTITY column that specifies the maximum value of the cycle range. It cannot be greater than the maximum value of the data type of the IDENTITY column. It must be greater than the INCREMENT BY value. If NO MAXVALUE is specified or this option is omitted, the maximum value is the maximum value of the data type of the IDENTITY column.

`MINVALUE` *`signed-numeric-literal`* `| NO MINVALUE`

A non-negative value that specifies the minimum value of the cycle range. If NO MINVALUE is specified or if this option is omitted, the minimum value is the minimum value of the data type of the IDENTITY column.

`NO CYCLE`

means that when the MAXVALUE maximum is reached for the IDENTITY column, an error is raised that indicates the maximum has been exceeded. Values will not be restarted. If an artificially low cycle range was created by setting the MAXVALUE value lower than the natural maximum allowed for the data type, you can use the ALTER TABLE ALTER COLUMN SET MAXVALUE option to raise the maximum value up to the natural maximum allowed for the data type. Raising the MAXVALUE allows more available values in the cycle range for the internal Sequence Generator. Inserts will be successful until the new MAXVALUE is reached. For more information, see the Syntax Description of ALTER TABLE on page 2-21.

`sg-location`

specifies a volume and optionally the node, subvolume, and filename for the SG Table. See SG Table on page 2-130.

HEADING '*heading-string*'| NO HEADING

specifies a string *heading-string* of 0 to 128 characters to use as a heading for the column if it is displayed with a SELECT statement in MXCI. The heading can contain characters only from the ISO88591 character set. The default heading is the column name. If you specify a heading that is identical to the column name, INVOKE and SHOWDDL do not display that heading.

If you specify NO HEADING or HEADING ‘’, NonStop SQL/MX stores this as HEADING ‘’, and the column name is displayed as the heading in a SELECT statement. The behavior for HEADING ‘’ is different from that of NonStop SQL/MP, which does not display anything for a heading in a SELECT statement if the heading is specified as HEADING ‘’.

CONSTRAINT *constraint*

specifies a name for the column or table constraint. *constraint* must have the same catalog and schema as *table* and must be unique among constraint names in its schema. If you omit the catalog portion or the catalog and schema portions of the name you specify in *constraint*, NonStop SQL/MX expands the constraint name by using the catalog and schema for *table*. See Database Object Names on page 6-13.

If you do not specify a constraint name, NonStop SQL/MX constructs an SQL identifier as the name for the constraint in the catalog and schema for *table.* The identifier consists of the fully qualified table name concatenated with a system-generated unique identifier. For example, a constraint on table A.B.C might be assigned a name such as A.B.C_123..._01... . Use the SHOWDDL statement to display this generated constraint name. See SHOWDDL Command on page 4-83.

NOT NULL [[NOT] DROPPABLE]

is a column constraint that specifies that the column cannot contain nulls. If you omit NOT NULL, nulls are allowed in the column. If you specify both NOT NULL and NO DEFAULT, each row inserted in the table must include a value for the column. See MXCI Parameters on page 6-77 and Null on page 6-80.

DROPPABLE specifies that you can drop the NOT NULL constraint by using ALTER TABLE at some later time. Dropping NOT NULL requires that you know the name of the constraint, either by using the CONSTRAINT *constraint* clause when the table is created or by using SHOWDDL to display the constraint name.

NOT DROPPABLE specifies that the NOT NULL constraint is permanent. Less space is required to store a column if the column has a permanent NOT NULL constraint, and updates and inserts are faster.

If the NOT NULL constraint does not include the [NOT] DROPPABLE clause, the value of the NOT_NULL_CONSTRAINT_DROPPABLE_OPTION attribute in the SYSTEM_DEFAULTS table is the default value for [NOT] DROPPABLE. If that attribute does not exist in the SYSTEM_DEFAULTS table, DROPPABLE is used.

Use the SHOWDDL statement to display the default that was used. See System Defaults Table on page 10-37.

`UNIQUE`
or
`UNIQUE (`*`column-list`*`)`

is a column or table constraint (respectively) that specifies that the column or set of columns cannot contain more than one occurrence of the same value or set of values. If you omit UNIQUE, duplicate values are allowed.

`column-list` cannot include more than one occurrence of the same column. In addition, the set of columns you specify on a UNIQUE constraint cannot match the set of columns on any other UNIQUE constraint for the table or on the PRIMARY KEY constraint for the table. All columns defined as unique must be specified as NOT NULL.

The maximum combined length of the columns depends on the block size of the index that supports the constraint. For 4K blocks, the maximum length is 2010 bytes and for 32K blocks, it is 2048 bytes.

`PRIMARY KEY [ASC[ENDING] | DESC[ENDING]] [[NOT] DROPPABLE]`
or
`PRIMARY KEY (`*`key-column-list`*`) [[NOT] DROPPABLE]`

is a column or table constraint (respectively) that specifies a column or set of columns as the primary key for the table. *`key-column-list`* cannot include more than one occurrence of the same column. In addition, the set of columns you specify on a PRIMARY KEY constraint cannot match the set of columns on any UNIQUE constraint for the table.

ASCENDING and DESCENDING specify the direction for entries in one column within the key. The default is ASCENDING.

The PRIMARY KEY value in each row of the table must be unique within the table. Columns within a PRIMARY KEY cannot contain nulls. A PRIMARY KEY defined for a set of columns implies that the column values are unique and not null. You can specify PRIMARY KEY only once on any CREATE TABLE statement.

DROPPABLE specifies that you can drop the PRIMARY KEY constraint with an ALTER TABLE statement at some later time. NOT DROPPABLE specifies that the PRIMARY KEY constraint is permanent. A PRIMARY KEY constraint is implemented more efficiently if the constraint is permanent. A SYSKEY is not generated for a table that has a NOT DROPPABLE PRIMARY KEY.

For a NOT DROPPABLE PRIMARY KEY, the maximum combined length of the columns depends on the block size of the table. For a DROPPABLE PRIMARY KEY, the maximum combined length of the columns depends on the block size of the supporting index. For both a DROPPABLE and NOT DROPPABLE PRIMARY KEYs, the maximum length is 2010 bytes for 4K blocks and 2048 bytes for 32K blocks.

For a PRIMARY, CLUSTERING, or UNIQUE key, the maximum number of key columns is 1024.

When a UNIQUE or PRIMARY KEY constraint is created on a table, all the constraint columns must have a NOT NULL clause in the CREATE TABLE statement.

If the PRIMARY KEY constraint does not include the [NOT] DROPPABLE clause and the STORE BY PRIMARY KEY clause does not appear in the table definition, the value of the PRIMARY_KEY_CONSTRAINT_DROPPABLE_OPTION attribute in the SYSTEM_DEFAULTS table is the default value. If that attribute does not exist in the SYSTEM_DEFAULTS table, NOT DROPPABLE is used. Use the SHOWDDL statement to display the default that was used. If the STORE BY PRIMARY KEY clause appears in the table definition, the PRIMARY KEY constraint is NOT DROPPABLE regardless of the value of the attribute.

If the PRIMARY KEY constraint does not include the [NOT] DROPPABLE clause and the STORE BY PRIMARY KEY clause appears in the table definition, and you make your PRIMARY KEY droppable, NonStop SQL/MX reports an error.

When possible, NonStop SQL/MX uses the primary key as the clustering key of the table in order to avoid creating a separate, unique index to implement the primary key constraint.

NonStop SQL/MX cannot implement the primary key as the clustering key if any of the following are true:

- You enter an explicit STORE BY clause, specifying a different set of columns than those specified for the primary key.
- You do not specify a PRIMARY KEY constraint within the CREATE TABLE statement.
- The PRIMARY KEY defined in the CREATE TABLE statement is droppable.

In any of these cases, NonStop SQL/MX implements the PRIMARY KEY as a separate unique index.

If an explicit STORE BY clause is used, NonStop SQL/MX does not allow the primary key constraint to have the NOT DROPPABLE clause. A PRIMARY KEY which is implemented by a separate unique index is always droppable.

Table 2-1 lists the maximum key size with respect to the block size.

Table 2-1. Maximum Key Sizes Available

DP2 block size	Max key size without triggers	Max key size available with triggers	Max # of key columns
4096	2010	1994	1024
32768	2048	2032	1024

The actual limit for primary keys, indexes, and clustering keys depends on the key specification, and will be at most the maximum key size limit.

`CHECK (`*`condition`*`)`

is a constraint that specifies a condition that must be satisfied for each row in the table. See Search Condition on page 6-108.

NonStop SQL/MX checks the condition whenever an operation occurs that might affect its value. The operation is allowed if the predicate in the condition evaluates to TRUE or null but prohibited if the predicate evaluates to FALSE.

You cannot refer to the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP function in a CHECK constraint, and you cannot use subqueries in a CHECK constraint. CHECK constraints cannot contain non-ISO88591 string literals.

`REFERENCES` *`ref-spec`*

specifies a references column constraint. The maximum combined length of the columns for a REFERENCES constraint depends on the block size of the supporting index. For 4K blocks, the maximum length is 2010 bytes and for 32K blocks, it is 2048 bytes.

`FOREIGN KEY (`*`column-list`*`) REFERENCES` *`ref-spec`*

is a column or table constraint (respectively) that specifies a referential constraint for the table, declaring that a column or set of columns (called a foreign key) in *`table`* can contain only values that match those in a column or set of columns in the table specified in the REFERENCES clause.

The two columns or sets of columns must have the same characteristics (data type, length, scale, precision), and there must be a UNIQUE or PRIMARY KEY constraint on the column or set of columns specified in the REFERENCES clause.

Without the FOREIGN KEY clause, the foreign key in *`table`* is the column being defined; with the FOREIGN KEY clause, the foreign key is the column or set of columns specified in the FOREIGN KEY clause.

`ref-spec` is:

referenced-table [(*column-list*)] [*referential triggered action*]

referenced-table is the table referenced by the foreign key in a referential constraint. *referenced-table* cannot be a view. *referenced-table* cannot be the same as *table*.

column-list specifies the column or set of columns in *referenced-table* that corresponds to the foreign key in *table*. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column list associated with FOREIGN KEY. If *column-list* is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

update rule specifies what *referential action* is taken when *column-list* in *referenced-table* is updated. If no ON UPDATE clause is specified, a default of ON UPDATE NO ACTION is assumed.

delete rule specifies what *referential action* is taken when a row in *referenced-table* is deleted. If no ON DELETE clause is specified, a default of ON DELETE NO ACTION is assumed.

referential action

RESTRICT *referential action* means that the referential check is made for each row. An error is raised when the referential constraint is violated.

ANSI SQL-99 standard: NO ACTION *referential action* means that the referential check is made at the end of the SQL statement. An error is raised when the referential constraint is violated.

NonStop SQL/MX does not support NO ACTION referential action in the way it is specified by ANSI SQL-99. However, you can change NO ACTION's behavior to be the same as RESTRICT by setting an appropriate value for the Control Query Default REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT. Options for this attribute are:

OFF	SQL issues an error.
SYSTEM	SQL issues warning 1302, indicating that it will behave like RESTRICT. This is the default value.
ON	Makes NO ACTION behave like RESTRICT, without warnings or errors.

When CASCADE is specified with the ON DELETE referential triggered action, a row in the referencing table and its corresponding row in the *referenced-table* is deleted. This maintains consistency between the referencing and referenced tables.

When SET NULL is specified with the ON DELETE referential triggered action, and a row from the referencing table matches the row in the *referenced-table*, the referencing column(s) of the referencing row from the referencing table is set to NULL.

When SET DEFAULT is specified with the ON DELETE referential triggered action, and a row from the referencing table matches the row in the *referenced-table*, the referencing column(s) of the referencing row from the referencing table is set to its DEFAULT value.

When CASCADE is specified with the ON UPDATE referential triggered action, a row in the referencing table and its corresponding row in the *referenced-table* is updated.

When SET NULL is specified with the ON UPDATE referential triggered action, and a row in the referencing table matches the row in the *referenced-table*, the referencing column(s) of the referencing row from the referencing table is set to NULL.

When SET DEFAULT is specified with the ON UPDATE referential triggered action, and a row in the referencing table matches the row in the *referenced-table*, the referencing column(s) of the referencing row from the referencing table is set to its DEFAULT value.

Note. The referential actions CASCADE, SET NULL, and SET DEFAULT are available only on systems running J06.09 and later J-series RVUs and H06.20 and later H-series RVUs.

---

referenced-table is the table referenced by the foreign key in a referential constraint. *referenced-table* cannot be a view. *referenced-table* cannot be the same as *table*.

column-list specifies the column or set of columns in *referenced-table* that corresponds to the foreign key in *table*. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column list associated with FOREIGN KEY. If *column-list* is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

A table can have an unlimited number of referential constraints, and you can specify the same foreign key in more than one referential constraint, but you must define each referential constraint separately. You cannot create self-referencing foreign key constraints.

Publish/Subscribe's embedded update and embedded delete statements are not allowed on tables with referential integrity constraints:

`STORE BY` *`store-option`*

specifies a set of columns on which to base the clustering key. The clustering key determines the order of rows within the physical file that holds the table. The storage order has an effect on how you can partition the object.

store-option is defined as:

PRIMARY KEY

bases the clustering key on the primary key columns. This store option requires that the primary key is NOT DROPPABLE. If the primary key is defined as DROPPABLE, NonStop SQL/MX returns an error.

key-column-list

bases the clustering key on the columns in the *key-column-list*. The key columns in *key-column-list* must be specified as NOT NULL NOT DROPPABLE. It cannot have a combined length more than 2002 bytes for 4K blocks and 2040 bytes for 32K blocks.

The default is PRIMARY KEY if you specified a PRIMARY KEY clause that has the NOT DROPPABLE constraint in the CREATE TABLE statement.

If you omit the STORE BY clause and you do not specify a PRIMARY KEY that has the NOT DROPPABLE constraint, the storage order is determined only by the SYSKEY. You cannot partition a table stored only by SYSKEY. See SYSKEYs on page 6-63.

LOCATION [*node.*]*$vo*lume[.*subvolume*.*file-name*]
[NAME *partition-name*]

specifies a physical location for the primary partition of the table.

node

is the name of a node on the Expand network.

For Guardian files representing a table or index partition or a view label, *node* can be any node from which the object's catalog is visible.

volume

is the name of an audited, non-SMF DAM volume on the specified node (or the Guardian volume named in the =_DEFAULTS define if none is specified).

If you do not specify a LOCATION clause and your system does not have a value for the DDL_DEFAULT_LOCATIONS default (either in your environment or at the system level) and your environment does not have a =_DEFAULTS value, the CREATE statement will fail with an error.

subvolume

must be the name of the schema subvolume for the schema in which the table is being created.

Follow these guidelines when using SQL/MX subvolume names:

- The name must begin with the letters *ZSD*, followed by a letter, not a digit (for example, ZSDa, not ZSD2).
- The name must be exactly eight characters long.
- The subvolume name you specify must match the designated schema subvolume name for the schema in which the object is being created. Otherwise, NonStop SQL/MX returns an error.

file-name

is a Guardian file name. *file-name* must be eight characters in length and must end with the digits `00` (zero zero).

partition-name

is an SQL identifier for a partition.

partn-file-option is:

```
{[RANGE] PARTITION
   [BY (partitioning-column  [,partitioning-column]...)]
   [(ADD range-partn-defn [,ADD range-partn-defn]...)]
```

defines secondary partitions for a range partitioned table.

BY (*partitioning-column* [,*partitioning-column*]...)

specifies the partitioning columns. The default is the default partitioning key created by the STORE BY clause. Partitioning character columns must derive from the ISO88591 character set. Partitioning columns cannot be floating-point data columns.

```
| HASH PARTITION
   [BY (partitioning-column [,partitioning-column]...)]
   [(ADD partn-defn [,ADD partn-defn]...)]}
```

defines secondary partitions for a hash partitioned table.

BY (*partitioning-column* [,*partitioning-column*]...)

specifies the columns that make up the partitioning key. If you do not specify this clause, the partitioning key is the same as the clustering key of the table. Partitioning columns cannot be floating-point data columns.

ADD *range-partn-defn*

defines a single secondary partition and includes the FIRST KEY and a *partn-defn*.

range-partn-defn is:

```
FIRST KEY {col-value | (col-value [,col-value]...)}
  partn-defn
```

specifies the beginning of the range for a range partitioned table. The FIRST KEY clause specifies the lowest values in the partition for columns stored in ascending order and the highest values in the partition for columns stored in descending order. These column values are referred to as the *partitioning key*.

col-value is a literal that specifies the first value allowed in the associated partition for that column of the partitioning key. If there are more storage key columns than *col-value* items, the first key value for each remaining key column is the lowest or highest value for the data type of the column (the lowest value for an ascending column and the highest value for a descending column). *col-value* must contain characters only from the ISO88591 character set.

If the table has a system-generated SYSKEY, its column list cannot consist only of column SYSKEY. The SYSKEY must be the last column of the column list, and you cannot specify a FIRST KEY value for the SYSKEY column. This limitation does not apply to a user-created SYSKEY column.

ADD *partn-defn*

defines a single secondary partition and includes the LOCATION of the partition.

partn-defn is:

```
LOCATION [\node.]$volume[.subvolume.file-name]
[EXTENT ext-size | (pri-ext-size [,sec-ext-size])]
[MAXEXTENTS num-extents]
    [NAME partition-name]
```

specifies a volume and optionally the node, subvolume, and filename for the partition.

node

is the name of a node on the Expand network. For Guardian files representing a table or index partition or a view label, *node* can be any node from which the object's catalog is visible.

volume

is the name of an audited, non-SMF DAM volume on the specified node (or the Guardian volume named in the =_DEFAULTS define if none is specified). If you do not specify a LOCATION clause, NonStop SQL/MX uses the default volume named in the =_DEFAULTS define.

If you do not specify a LOCATION clause and your system does not have a value for the DDL_DEFAULT_LOCATIONS default (either in your environment or at the system level) and your environment does not have a =_DEFAULTS value, the CREATE statement will fail with an error.

You can locate more than one partition of a table on a single disk volume.

subvolume

is the name of the schema subvolume for the schema in which the table is being created.

Follow these guidelines when using SQL/MX subvolume names:

- The name must begin with the letters *ZSD*, followed by a letter, not a digit (for example, ZSDa, not ZSD2).
- The name must be exactly eight characters long.
- The subvolume name you specify must match the designated schema subvolume name for the schema in which the object is being created. Otherwise, NonStop SQL/MX returns an error.

file-name

is a Guardian file name. *file-name* names must be eight characters long and must end with the digits "00" (zero zero).

partition-name

is an SQL identifier for a partition.

partn-file-option is an SQL/MX extension.

See PARTITION Clause on page 7-6.

ATTRIBUTE[S] *attribute* [,*attribute*]...

specifies attributes of the physical file that holds the table. In an ATTRIBUTES clause that is within a PARTITION clause, you must separate *attributes* with a space. In ATTRIBUTES clauses in other places, you can separate *attributes* with either a space or a comma. You can specify these file attributes:

ALLOCATE/DEALLOCATE on page 9-2	Controls amount of disk space allocated.
AUDITCOMPRESS on page 9-3	Controls whether unchanged columns are included in audit records.
BLOCKSIZE on page 9-4	Sets size of data blocks.
CLEARONPURGE on page 9-5	Controls disk erasure when table is dropped.
EXTENT on page 9-6	Controls size of extents that are allocated on disk.
MAXEXTENTS on page 9-7	Controls maximum disk space to be allocated.

If you use the LIKE specification and you do not specify `ATTRIBUTE[S]` *`attribute`* `[,`*`attribute`*`]...`, NonStop SQL/MX uses the attributes associated with the *`source-table`*.

For more information, see the entry for a specific attribute.

`LIKE` *`source-table`* `[`*`include-option`*`]...`

directs NonStop SQL/MX to create a table like the existing table, *`source-table`*, omitting constraints (with the exception of the NOT NULL and NOT DROPPABLE PRIMARY KEY constraints), headings, and partitions unless *`include-option`* clauses are specified. The *`source-table`* must be the ANSI name of an SQL/MX format table (you cannot specify an SQL/MP table).

`ATTRIBUTE[S]` *`attribute`* `[,`*`attribute`*`]...` and the STORE BY *`store-option`* are copied from the *`source-table`* if they are not explicitly specified as file options following the LIKE specification.

The *`include-option`* clauses are specified as:

`WITH CONSTRAINTS`

directs NonStop SQL/MX to use constraints from *`source-table`*. Constraint names for *`table`* are randomly generated unique names. NonStop SQL/MX does not include FOREIGN KEY table constraints or REFERENCES column constraints.

This table show the results of specifying or not specifying WITH CONSTRAINTS on primary key constraints:

WITH CONSTRAINTS clause?	**Source table has...**	**Target table will have...**
No	DROPPABLE primary key constraint	No primary key constraint.
Yes	DROPPABLE primary key constraint	The same DROPPABLE primary key constraint with a different name.
With or without	NOT DROPPABLE primary key constraint	The same NOT DROPPABLE primary key constraint with a different name.

When you perform a CREATE TABLE LIKE, whether or not you include the WITH CONSTRAINTS clause, the target table will have all the NOT NULL DROPPABLE column constraints that exist for the source table, plus all the NOT NULL NOT DROPPABLE column constraints that exist for the source table. They will have different constraint names.

`WITH HEADINGS`

directs NonStop SQL/MX to use column headings from *`source-table`*.

`WITH PARTITIONS`

directs NonStop SQL/MX to use partition definitions from *`source-table`*. Each new table partition resides on the same volume as its original *`source-table`* counterpart. The new table partitions do not inherit partition names from the original table. Instead, NonStop SQL/MX generates new names based on the physical file location.

If you specify the LIKE clause and the PARTITION *`file-option`*, you cannot specify `WITH PARTITIONS`. If you specify the LIKE clause and the STORE BY *`store-option`*, you cannot specify `WITH PARTITIONS`. If the *`source-table`* has a partitioned index for a constraint, an index is created for the constraint on the target table, with attributes that differ from the attributes of the source table's index.

Considerations for CREATE TABLE

Reserved Table Names

Table names prefixed by the name of a UMD table are reserved. You cannot create tables with such names. For example, you cannot create a table named HISTOGRAMS_MYCOPY.

Partitions

If there is a possibility that you might need to partition a table in the future, you should create it with at least one partition. This avoids recompilation if you add more partitions later.

The LIKE specification

The CREATE TABLE LIKE statement does not create views, owner information, or privileges for the new table based on the source table. Privileges associated with a new table created by using the LIKE specification are defined as if the new table is created explicitly by the current user.

The existing behavior of CREATE TABLE LIKE is retained. CREATE TABLE LIKE does not create the RI constraint for the target table.

If the source table has any unique or droppable primary key constraints, NonStop SQL/MX creates indexes for them on the target table. Other indexes on the source table are not created on the target table.

The LIKE specification ignores triggers.

CREATE TABLE LIKE with an IDENTITY column is supported. For a table with an IDENTITY column, the target table inherits the IDENTITY property of a column along with the corresponding sequence generating attributes from the source table.

Storage Order and the LIKE Specification

The STORE BY clause determines the storage order of the records in the new table:

`STORE BY PRIMARY KEY`	The new table is ordered by the primary key of the source table.
`STORE BY` *`key-column-list`*	The new table is ordered by the new *`key-column-list.`*
No `STORE BY` clause	The new table is ordered by the storage key of the source table.

Audited and Nonaudited Tables

NonStop SQL/MX does not support nonaudited SQL/MX tables, but scenarios exist that require nonaudited tables. For example, suppose that you want updates to occur even if the operation is rolled back for logging purposes. In this case, you should use NonStop SQL/MP to create a nonaudited SQL/MP table.

Authorization and Availability Requirements

To create a table, you must own its schema or be the super ID.

To create a constraint on the table that refers to a column in another table, you must have REFERENCES privileges on that column and access to the table that contains the column. If the constraint refers to the other table in a query expression, you must also have SELECT privileges on the other table.

Reduced Space Requirements for NOT DROPPABLE Constraints

Using the NOT DROPPABLE option on a NOT NULL constraint reduces the space required for the table. A column that allows nulls—or that might allow nulls at some later time—uses two extra bytes in each row to store the null indicator. If you specify that the NOT NULL constraint is NOT DROPPABLE, NonStop SQL/MX creates the table without these extra bytes.

Using the NOT DROPPABLE option on a PRIMARY KEY or using STORE BY PRIMARY KEY reduces the space required for the table and eliminates the need to create an index for accessing the table by primary key.

Constraints Implemented With Indexes

NonStop SQL/MX uses indexes to implement all UNIQUE constraints, the foreign key portion of all referential constraints, and any PRIMARY KEY constraints that are not enforced by the clustering key. Necessary indexes are automatically created when you create a table with these constraints. If you add a constraint to an existing table, NonStop SQL/MX checks if an existing index can be used to implement the constraint, creating a new index (if possible, with the same name as the constraint) if needed.

For small tables, you need not be concerned about the details of this mechanism. For large tables, however, the indexes used to enforce constraints can require significant amounts of disk space. You might prefer to create and partition constraint-supporting indexes directly so that you can control the use of disk space or so that you can specify indexes that provide more effective access paths for your application than those created by default to support constraints.

To create constraint-supporting indexes directly, use CREATE TABLE to create the table without index-implemented constraints, use CREATE INDEX to create appropriate indexes, and then use ALTER TABLE ... ADD CONSTRAINT to add constraints to the table.

Limits for Tables

The maximum size of a row depends on the block size as described in Table 2-2.

If the table has a SYSKEY, the SYSKEY column requires 8 bytes. The number of bytes required to store a column depends on the data type of that column. If the column is nullable, NonStop SQL/MX uses an additional 2 bytes for the NULL indicator. Each variable-length character column uses an additional 8 bytes for the column length. There can be a maximum of 2100 columns in a row.

Table 2-2. Maximum Row Sizes Available

DP2 block size	Max row size available to users	Max # of Columns
4096	4036	2100
32768	32708	2100

Tables and Triggers

The primary key length for a table with triggers cannot exceed 2032 bytes. A table that does not have triggers can have a primary key of 2048 bytes. For information about this limit, see Triggers and Primary Keys on page 2-150.

Creating Partitions Automatically

NonStop SQL/MX uses Partition Overlay Specification (POS) so that MXCI, MXCS, JDBC T4, and JDBC T2 users can automatically create hash-partitioned tables with the CREATE TABLE statement. NonStop SQL/MX does not support automatic creation of range-partitioned tables.

Applications can control whether POS is enabled, the number of partitions, and the physical location of the partitions.

The following CONTROL QUERY DEFAULT attributes determine the physical location and the number of partitions:

- POS_LOCATIONS
- POS_NUM_OF_PARTNS

The POS_RAISE_ERROR attribute controls how errors are displayed. For values and syntax of these attributes, see Partition Management on page 10-63.

To enable POS, set the POS_NUM_OF_PARTNS attribute to a value greater than 1.

To activate POS, ensure that the following conditions are true:

- The POS feature is enabled during execution of a CREATE TABLE statement.
- The application that issues the CREATE TABLE DDL statement is an MXCS/JDBC session or an MXCI session.
- The CREATE TABLE statement does not specify an `add location` using the partitioning syntax.
- The CREATE TABLE statement specifies either the PRIMARY KEY or the STORE BY clause.

If you specify the LOCATION clause for the primary partition, the partition resides on the volume specified in that clause and not in the location specified in POS_LOCATIONS. If the LOCATION clause is not specified, the primary partition location will be picked at random among those specified in POS_LOCATIONS. It will not be the first location specified in POS_LOCATIONS.

If you do not specify the LOCATION clause and if you set POS_LOCATIONS, the primary partition resides on the first volume specified in POS_LOCATIONS. The other partitions reside on the volumes you specify in POS_LOCATIONS in a round-robin fashion.

If the LOCATION clause is not specified in the CREATE TABLE statement and the POS_LOCATIONS CQD is empty, NonStop SQL/MX randomly selects the location(s) from the full set of audited volumes.

These examples show how partitions are created automatically using combinations of attribute values:

1. Specify POS_NUM_OF_PARTNS as 3 and list three locations in POS_LOCATIONS: $VOL1, $VOL2, and $VOL3.

   NonStop SQL/MX will place the primary partition on $VOL1, the second partition on $VOL2, and the third partition on $VOL3.

2. Specify POS_NUM_OF_PARTNS as 5 and list three locations in POS_LOCATIONS: $VOL1, $VOL2, and $VOL3.

   NonStop SQL/MX will place the primary partition on $VOL1, the second partition on $VOL2, the third partition on $VOL3, the fourth on $VOL1, and the fifth on $VOL2.

3. Specify POS_NUM_OF_PARTITIONS as 4, and list three locations in POS_LOCATIONS: $VOL1, $VOL2, and $VOL3. In addition, include a LOCATION clause in the CREATE statement that specifies $DATA1.

NonStop SQL/MX will place the primary partition on $DATA1, the second partition on $VOL1, the third partition on $VOL2, and the fourth partition on $VOL3.

Creating a Table Without STORE BY Clause or Primary Key

NonStop SQL/MX bases table partitioning on clustering key columns, specified by the STORE BY clause or, if there is no STORE BY clause, the primary key columns. If you do not specify the STORE BY or PRIMARY KEY columns on a table, NonStop SQL/MX cannot partition the table. If you attempt to use POS with such a table, you will not receive an error. POS creates a nonpartitioned table in the same way that NonStop SQL/MX creates a nonpartitioned table without the LOCATION clause as part of the CREATE TABLE statement. The location of this table is not based on POS_LOCATIONS or automatic disk location.

Partitioning Columns

Use the PARTITION BY clause to decouple the partitioning key from the clustering key. Without the PARTITION BY clause, the partitioning columns of the table are same as the clustering key columns. POS can be used to create partitions automatically for tables with decoupled keys.

IDENTITY Column and internal Sequence Generators

An IDENTITY column is a numeric column in a table for which the system generates unique values using an internal Sequence Generator.

ANSI SQL Standard 2003 suggests two ways of generating unique numeric values:

1. Internal Sequence Generator: An internal Sequence Generator is implicitly created when an IDENTITY column is defined and is associated only with that IDENTITY column.
2. External Sequence Generator: An external sequence generator is explicitly created using the CREATE SEQUENCE statement. The external sequence generator is a schema level database object that the application uses to generate values for a numeric column. The values generated by the external sequence generator are unique for that sequence generator and can be used to create unique values across a set of tables in a schema.

Sequence generators contain the following attributes called SG Attributes:

- a data type
- a minimum value
- a maximum value
- a start value
- an increment
- a cycle option

See CREATE TABLE Statement on page 2-107 for more information.

SG Table

When an IDENTITY column is defined, a sequence generator table (SG Table) is created. The SG Table contains a column, `CURRENT_VALUE`. The `CURRENT_VALUE` represents the next available value for the IDENTITY column. Each SG Table contains only one row, with a primary key value of zero. The name space and object type for the table has a value SG.

Table 2-3. SG Table for IDENTITY COLUMN

Column Number	Column Name	Data Type	Description
*1	ZERO_PKCOL	LARGEINT	Primary key with value zero
2	CURRENT_VALUE	LARGEINT	Contains the current value of the IDENTITY column

* Indicates primary key

Use the *`sg-location`* clause during CREATE TABLE to specify a location for the SG Table. If you do not specify a location, the system default location is used. Since an SG Table has base table characteristics, it includes an entry in the metadata OBJECTS, ACCESS_PATHS, COLS, and PARTITIONS tables, etc.

The SG Table is updated upon a request to obtain the next value from the sequence generator. The `CURRENT_VALUE` from the SG Table is selected and returned as the next value. Then, the `CURRENT_VALUE` is updated with `CURRENT_VALUE` plus INCREMENT BY value specified in SG Attributes.

A new independent transaction different from the user transaction is started and committed for the SG Table select and update operations. A separate ESP process houses the operator that starts and commits this new transaction. To reduce the impact from this overhead, see Sequence Generator Cache on page 2-134.

Considerations for an IDENTITY column

- If you define the IDENTITY column as GENERATED BY DEFAULT AS IDENTITY, values for the IDENTITY column are generated by default. If you specify a value for the IDENTITY column, NonStop SQL/MX uses that value and does not generate a unique value for that row.
- An IDENTITY column can be the primary key or part of the primary key.
- An IDENTITY column can be the partitioning key or part of the partitioning key.
- An IDENTITY column can be the clustering key or part of the clustering key.
- An IDENTITY column can be defined on a HASH or a RANGE partitioned table.
- An IDENTITY column can be part of an INDEX.

- An IDENTITY column can be part of CHECK and RI constraints.
- You can alter the MAXVALUE and INCREMENT BY sequence generator options of an IDENTITY column using the ALTER TABLE ALTER COLUMN statement. See Altering the MAXVALUE and INCREMENT BY options on IDENTITY columns on page 2-34.
- INSERT...SELECT operations on tables with an IDENTITY column is supported.
- CREATE TABLE LIKE on a source table with an IDENTITY column is supported. The target table will inherit the source table column and sequence generator attributes.
- When a table with an IDENTITY column is dropped, the internal Sequence Generator is dropped. Metadata entries for the internal Sequence Generator are cleared and the SG Table associated with the IDENTITY column is dropped. See SG Table on page 2-130.

Restrictions for an IDENTITY Column

- At most one IDENTITY column can be specified in a table.
- The IDENTITY column cannot have a NOT NULL NOT DROPPABLE constraint. If not specified, the system will add the constraint.
- An IDENTITY column definition supports the following data types only: `LARGEINT`, `unsigned INTEGER`, and `unsigned SMALLINT` data types.
- The NO CYCLE option is the only cycle option supported.
- Only ascending IDENTITY values are supported. IDENTITY values are called ascending if the increment value is a non-negative value.
- These options can be specified one time only for each table:
  - START WITH
  - INCREMENT BY
  - MAXVALUE | NO MAXVALUE
  - MINVALUE | NO MINVALUE
  - NO CYCLE
- The START WITH value must be less than or equal to the MAXVALUE and greater than or equal to the MINVALUE. If the START WITH option is not specified, the start value is the MINIMUM value.
- The INCREMENT BY option cannot be 0 (zero), less than 0 (zero), and cannot be greater than the maximum value of the data type of the IDENTITY column.
- If the MAXVALUE option is not specified, or if NO MAXVALUE is specified, the maximum value is the maximum value of the data type of the IDENTITY column.

The MAXVALUE option value must be greater than the value of the MINVALUE and a valid numeric value for the IDENTITY column data type.

- If the MINVALUE option is not specified, or if NO MINVALUE is specified, then the minimum value is the minimum value of the data type of the IDENTITY column. In the case of the IDENTITY column being of data type LARGEINT, the minimum default value will be zero, not -9223372036854775808.
- Mixed DML and DDL operations performed under the same user transaction are not supported for an INSERT operation that contains an IDENTITY column. TMF Error 73 can occur because the DDL and DML operations share the same user transaction on a table lock on the SG Table.
- For a table with only one column, which is an IDENTITY column, the tuple list of an INSERT statement cannot consist of only DEFAULT values. You must specify the input values, otherwise error 3431 will be raised. For example, this statement issues error 3431:

```
INSERT INTO t1 VALUES (DEFAULT), (DEFAULT);
```

- You cannot add an IDENTITY column by using the ALTER TABLE statement.
- Expressions involving the keyword DEFAULT are not allowed as IDENTITY column values. You must specify the keyword DEFAULT or supply a valid value. Error 3411 will be raised if an expression is specified for an IDENTITY column value. For example, this statement raises error 3411 indicating that an expression, DEFAULT+15 is used as a value for IDENTITY column, assuming that the first column is an IDENTITY column.

```
INSERT INTO t1 VALUES (DEFAULT+15, 45);
```

- UPDATE operations on IDENTITY columns defined as GENERATED ALWAYS AS IDENTITY are not allowed.
- For IDENTITY columns defined as type LARGEINT, the maximum value is 9223372036854775806, one less than the LARGEINT maximum.
- For an IDENTITY column, the tuple list cannot have mixed user and DEFAULT values specified. You must specify values for all tuples in the tuple list or specify DEFAULT for all tuples in the tuple list. For example, error 3414 is raised in the following case, assuming that the first column in the table `t_id_s` is an IDENTITY column. Notice that the third row contains a user specified value of '50' for the IDENTITY column and the other rows specify DEFAULT.

```
INSERT INTO t_id_s values (DEFAULT,’1’,1),
                          (DEFAULT,’2’,2),
                          (50,’3’,3),
                          (DEFAULT,’4’,4)
                            (DEFAULT,’5’,5);
```

Generating Values for an IDENTITY Column

You can use IDENTITY columns to automatically generate unique values. The generated values are unique across all partitions of the table for the IDENTITY column.

You can declare IDENTITY columns in the CREATE TABLE statement. IDENTITY columns can be used as surrogate keys. They can also be used to uniquely identify records with the same key.

Difference Between GENERATED ALWAYS AS IDENTITY and GENERATED BY DEFAULT AS IDENTITY

You can use GENERATED BY DEFAULT AS IDENTITY to allow both user-supplied and system-generated column values for the IDENTITY column. The GENERATED ALWAYS AS IDENTITY option provides system-generated unique values only. It does not allow user-supplied IDENTITY column values.

Generating the System-Generated Value for an IDENTITY Column

NonStop SQL/MX generates the next value for an IDENTITY column in these ways:

- You can specify DEFAULT as a value for the IDENTITY column in the INSERT statement, provided it is not the only column.

  For example: `INSERT INTO tbl1 (DEFAULT, 10);` assuming the first column is defined as an IDENTITY column.

- An INSERT statement specifying the columns to be inserted, but leaving out the IDENTITY column.

  For example: `INSERT INTO tbl1(b) values (10);` assuming that `tbl1` has an IDENTITY column and the IDENTITY column has been omitted from the column list.

Generating Unique Values for an IDENTITY Column

The sequence generator feature provides a method for generating unique values for an IDENTITY column. NonStop SQL/MX guarantees to generate unique values if the IDENTITY column is defined as GENERATED ALWAYS AS IDENTITY with the NO CYCLE option.

Available Values for an IDENTITY Column

The IDENTITY column can be defined as `signed LARGEINT, unsigned INTEGER,` and `unsigned SMALLINT`. Each data type has a natural maximum value. The settings chosen for START WITH, MINVALUE, MAXVALUE, and INCREMENT BY create a valid range of available numbers before a maximum is reached. The settings MAXVALUE and INCREMENT BY can be altered using ALTER TABLE ALTER COLUMN to change an artificially low range specified during the CREATE TABLE process. For more information, see Altering the MAXVALUE and INCREMENT BY options on IDENTITY columns on page 2-34.

Duplicate Values for an IDENTITY Column

Duplicates can be generated for an IDENTITY column if the IDENTITY column is defined as GENERATED BY DEFAULT AS IDENTITY and you specify a duplicate value for the column.

Sequence Generator Cache

The next values of the sequence generator can be cached in the operator that assigns the generated value to the IDENTITY column. Caching ability is provided to reduce the bottleneck at the SG table. The bottleneck can occur from many concurrent users getting the next value from the sequence generator table. For large insert operations, caching also reduces the overhead incurred from selecting and updating the sequence generator table while obtaining the next values.

The sequence generator cache is not a global cache; it resides in the instance that assigns the value for the IDENTITY column. The instance can either be a process (ESP or MASTER) within the same statement or from another application inserting values into the same table.

There are two types of sequence generator cache:

1. User-Defined cache: User-Defined cache value is specified by setting the Control Query Default (CQD) `SEQUENCE_GENERATOR_CACHE` at compile time. If this value is greater than its default of 0, then this is a pure override. Each cache size obtained will be for the `SEQUENCE_GENERATOR_CACHE` size and does not change.

2. Adaptive Cache: The cache value dynamically changes at runtime based on the load of the query. The following three CQD settings are used to manipulate the cache value dynamically at runtime. The computation starts with the `SEQUENCE_GENERATOR_CACHE_INITIAL` value, then multiplies it with the `SEQUENCE_GENERATOR_CACHE_INCREMENT` and compares it to the `SEQUENCE_GENERATOR_CACHE_MAXIMUM`. The adaptive cache is the default.

    Example: assume the following settings:

    `SEQUENCE_GENERATOR_CACHE_INITIAL` of “2”
    `SEQUENCE_GENERATOR_CACHE_INCREMENT` of “10”
    `SEQUENCE_GENERATOR_CACHE_MAXIMUM` of “20000”

    `SEQUENCE_GENERATOR_CACHE_INITIAL` provides the initial value of “2”. A cache of “2” is obtained. When the cache is exhausted, “2” is multiplied by the `SEQUENCE_GENERATOR_CACHE_INCREMENT` of “10”, supplying a new cache of “20” numbers. When the “20” values are exhausted, the current cache value of “20” is multiplied by `SEQUENCE_GENERATOR_CACHE_INCREMENT` “10”, supplying “20 X 10” = “200” new cache values. This dynamic calculation continues incrementing the numbers from “2” to “20” to “200” to “2000” to “20000”. “20000” is the `SEQUENCE_GENERATOR_CACHE_MAXIMUM`, so no further multiplication occurs. The maximum is used from that point on to determine the cache size.

    The default values for these CQDs are:

    `SEQUENCE_GENERATOR_CACHE_INITIAL` default of “1”
    `SEQUENCE_GENERATOR_CACHE_INCREMENT` default of “10”
    `SEQUENCE_GENERATOR_CACHE_MAXIMUM` default of “10000”

    Also, the SQL/MX Optimizer sets the `SEQUENCE_GENERATOR_CACHE_INITIAL` value based on the cardinality estimate of the number of rows and number of ESPs for the instance that assigns the generated value to the IDENTITY column value. SEQUENCE_GENERATOR_CACHE_INITIAL is equal to the cardinality estimate of the row divided by the number of ESPs.

The SQL/MX Optimizer setting can be overridden by setting non-default values for CQDs SEQUENCE_GENERATOR_CACHE_INITIAL and SEQUENCE_GENERATOR_CACHE_MAXIMUM. In other words, the default settings for these CQDs cause the optimizer to use the cardinality estimate and number of ESPs to set the SEQUENCE_GENERATOR_CACHE_INITIAL value.

Note. Adaptive-Cache is enabled only when the CQD value for SEQUENCE_GENERATOR_CACHE is 0 (zero).

It is highly recommended to use the User-Defined cache to gain familiarity with the IDENTITY column feature and then adapt the Adaptive Cache based on application needs. The CQD values for the Adaptive Cache must be carefully chosen by weighing the performance implications and the scalability requirements of the application.

Gaps in IDENTITY column values

The INSERT query plan that generates IDENTITY column values has the capability to cache next values depending on the number of rows in the INSERT statement. Caching ability is provided to reduce the bottleneck at the SG Table for parallel INSERT operations. Caching also reduces the overhead of fetching and updating the CURRENT_VALUE in the SG Table.

Gaps can occur in the sequence of the IDENTITY column values if the unused values in cache are lost. Unused cache values will be lost under the following scenarios:

- When an error occurs during an INSERT and transaction rollback occurs
- The process (ESP or MASTER Executor) housing the cache shuts down either as a result of query completion or from process failure
- System failures such as CPU halts
- If the internal transaction updating the SG Table was committed, but the INSERT with the user transaction fails.

It is recommended that the user choose a cache value that meets the performance needs of their application while minimizing the potential for large gaps in the sequence.

Gaps in sequence generator values

There can be gaps in sequence numbers acquired by a session if two sessions concurrently increment the same sequence. This is because sequence numbers are generated by different sessions and one session cannot get the sequence number generated by another session. After a sequence value is generated by a session, that session can continue to access the value even when the sequence is incremented or decremented by other sessions.

The gaps can also be generated because of caching next values as explained in Gaps in IDENTITY column values.

SQL/MX Extensions to CREATE TABLE

This statement is supported for compliance with ANSI SQL:1999 Entry Level. SQL/MX extensions to the CREATE TABLE statement are [NOT] DROPPABLE, ASCENDING, DESCENDING, STORE BY, LOCATION, PARTITION, ATTRIBUTE, and LIKE clauses.

Considerations for Referential Integrity

Circular Dependency

The following situations cause circular dependency when adding a Referential Integrity (RI)/Trigger:

- A situation where the UPDATE/DELETE/INSERT operations on the table being modified invoke RI(s)/trigger(s), thereby re-invoking the same RI/trigger with the same operation as the RI/trigger invoked earlier. This is an example of a circular dependency situation, which does not allow you to create this RI/trigger. **Exception**: If the circular dependency path consists of only triggers, the situation is not considered circular dependency for the reasons of backward compatibility.
- A situation where a few tables are interconnected by RIs, such that the referencing columns of one RI are the same as the referenced columns of another RI. This is another example of a circular dependency situation, which does not allow you to create this RI.

Conflicting and Duplicate Constraints

A referential integrity constraint that is created with new RI actions can conflict or be a duplicate of the already existing columns.

Conflicting Constraints

- The two constraints in a table conflict if the referenced table is the same and the referencing columns overlap or
- The two constraints of a table conflict if the referenced table and the referencing columns are the same and are in the same order, but the RI actions are different.

Duplicate Constraints

The two constraints of a table are said to be duplicate if the referenced table and the referencing columns are in the same order and the RI actions are the same.

If the existing RI actions for both the update and delete rule are NO ACTION/RESTRICT, and if the newly added RI constraint also has RI actions NO ACTION/RESTRICT for both the update and delete rule, they are not said to be duplicate or conflicting. This is to support backward compatibility.

Utilities

The utilities Backup/Restore, MXExportDDL/MXImportDDL, and NSM web support the newly added RI actions CASCADE/SET NULL/SET DEFAULT in addition to NO ACTION and RESTRICT.

The utilities DUP and PurgeData retain their existing behavior. The DUP utility does not support the RI constraints duplication and Purgedata does not allow you to purge data from a referred table.

Usage and Performance

The RI actions CASCADE, SET NULL, and SET DEFAULT enable you to maintain data integrity between tables. Performing RI actions is resource-intensive because indexes and multiple tables are involved, which can result in a significant drop in performance of queries when a large dataset is involved. Therefore, it is important that you consider the performance implication while defining RI relationships.

Examples of CREATE TABLE

- This example creates a table stored by primary key. The clustering key is the primary key.

```
CREATE TABLE SALES.ODETAIL
  ( ordernum     NUMERIC (6) UNSIGNED  NO DEFAULT  NOT NULL,
    partnum      NUMERIC (4) UNSIGNED  NO DEFAULT  NOT NULL,
    unit_price   NUMERIC (8,2)         NO DEFAULT  NOT NULL,
    qty_ordered  NUMERIC (5) UNSIGNED  NO DEFAULT  NOT NULL,
    PRIMARY KEY (ordernum, partnum) NOT DROPPABLE )
  STORE BY PRIMARY KEY;
```

- This example creates a table stored by the key column list. The clustering key is ordernum, partnum, SYSKEY.

```
CREATE TABLE SALES.ODETAIL
  ( ordernum     NUMERIC (6) UNSIGNED  NO DEFAULT  NOT NULL,
    partnum      NUMERIC (4) UNSIGNED  NO DEFAULT  NOT NULL,
    unit_price   NUMERIC (8,2)         NO DEFAULT  NOT NULL,
    qty_ordered  NUMERIC (5) UNSIGNED  NO DEFAULT  NOT NULL)
  STORE BY (ordernum, partnum);
```

- This example creates a table stored by the SYSKEY. The clustering key is the SYSKEY, type LARGEINT.

```
CREATE TABLE SALES.ODETAIL
  ( ordernum     NUMERIC (6) UNSIGNED  NO DEFAULT  NOT NULL,
    partnum      NUMERIC (4) UNSIGNED  NO DEFAULT  NOT NULL,
    unit_price   NUMERIC (8,2)         NO DEFAULT  NOT NULL,
    qty_ordered  NUMERIC (5) UNSIGNED  NO DEFAULT  NOT NULL)
  ;
```

- This example creates a table like the JOB table with the same constraints:

```
CREATE TABLE SAMDBCAT.PERSNL.JOB_CORPORATE
  LIKE SAMDBCAT.PERSNL.JOB WITH CONSTRAINTS;
```

- This example creates table `tab1` with partitions named `partition1` and `partition2`. It then creates table `tab2` like `tab1` with partitions. `tab2`'s partitions have different names than the partitions on `tab1`.

  - Create `tab1`:

```
>>Create table tab1
( a INT not null PRIMARY KEY, b INT)
range partition by (a)
(add first key 2   location $HIJO  NAME partition1 ,
 add first key 512 location $CHINA NAME partition2 )
attribute
extent (1024, 1024),
maxextents 16;

--- SQL operation complete.
```

  - Create `tab2`:

```
>>create table tab2 like tab1 with partitions;

--- SQL operation complete.
```

  - Perform SHOWDDL to display `tab1`'s properties:

```
>>showddl tab1;

CREATE TABLE J1.SCH1.TAB1
  (
    A               INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , B                                INT DEFAULT NULL
  , CONSTRAINT J1.SCH1.TAB1_264669593_1268 PRIMARY KEY
     (A ASC) NOT DROPPABLE
  , CONSTRAINT J1.SCH1.TAB1_535649593_1268 CHECK
       (J1.SCH1.TAB1.A IS NOT NULL)
      NOT DROPPABLE
  )
  LOCATION \CARNAG.$SARA.ZSDCL87P.DMP33T00
  NAME CARNAG_SARA_ZSDCL87P_DMP33T00
  ATTRIBUTES EXTENT (1024, 1024), MAXEXTENTS 16
  PARTITION
  (
    ADD FIRST KEY (2)
      LOCATION \CARNAG.$HIJO.ZSDCL87P.GHV33T00
      NAME PARTITION1
      EXTENT (1024, 1024) MAXEXTENTS 16
  , ADD FIRST KEY (512)
      LOCATION \CARNAG.$CHINA.ZSDCL87P.ZN133T00
      NAME PARTITION2
      EXTENT (1024, 1024) MAXEXTENTS 16
  )
  STORE BY (A ASC)
```

```
  ;

--- SQL operation complete.
```

- Perform SHOWDDL to display `tab2`'s properties. Note that the partitions are now named `CARNAG_HIJO_ZSDCL87P_PT245W00` and `CARNAG_CHINA_ZSDCL87P_S1645W00`:

```
>>showddl tab2;

CREATE TABLE J1.SCH1.TAB2
  (
    A                INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , B                                  INT DEFAULT NULL
  , CONSTRAINT J1.SCH1.TAB2_378461764_1268 PRIMARY KEY
     (A ASC) NOT DROPPABLE
  , CONSTRAINT J1.SCH1.TAB2_753441764_1268 CHECK
      (J1.SCH1.TAB2.A IS NOT NULL)
      NOT DROPPABLE
  )
  LOCATION \CARNAG.$SARA.ZSDCL87P.H5V45W00
  NAME CARNAG_SARA_ZSDCL87P_H5V45W00
  ATTRIBUTES EXTENT (1024, 1024), MAXEXTENTS 16
  PARTITION
  (
    ADD FIRST KEY (2)
      LOCATION \CARNAG.$HIJO.ZSDCL87P.PT245W00
      NAME CARNAG_HIJO_ZSDCL87P_PT245W00
      EXTENT (1024, 1024) MAXEXTENTS 16
  , ADD FIRST KEY (512)
      LOCATION \CARNAG.$CHINA.ZSDCL87P.S1645W00
      NAME CARNAG_CHINA_ZSDCL87P_S1645W00
      EXTENT (1024, 1024) MAXEXTENTS 16
  )
  STORE BY (A ASC)
  ;

--- SQL operation complete.
>>
```

- This example creates table `mytable` with hash partitions.

```
create table mytable
   ( col1 TIMESTAMP default current_timestamp not null
   , col2 INT not null
   , col3 VARCHAR (30)
   , col4 SMALLINT not null
   , PRIMARY KEY (col4, col1) )
location $VOL1
hash partition by (col4)
   ( add location $VOL2
   , add location $VOL3
   , add location $VOL4)
attribute
extent (1024, 1024),
maxextents 16
```

```
;

create unique index mytable_idx1 on mytable(col2, col1)
LOCATION $vol1
hash partition by (col2)
   ( add location $VOL2
   , add location $VOL3
   , add location $VOL4
   , add location $VOL5)
;
```

- This example creates a table stored by primary key. These defaults are in effect: `POS_LOCATIONS` is set to `$VOL1, $VOL2, $VOL3` and `POS_NUM_OF_PARTNS` is set to 3.

```
CREATE TABLE SALES.ODETAIL
  ( ordernum     NUMERIC (6) UNSIGNED  NO DEFAULT  NOT NULL,
    partnum      NUMERIC (4) UNSIGNED  NO DEFAULT  NOT NULL,
    unit_price   NUMERIC (8,2)         NO DEFAULT  NOT NULL,
    qty_ordered  NUMERIC (5) UNSIGNED  NO DEFAULT  NOT NULL,
    PRIMARY KEY (ordernum, partnum) NOT DROPPABLE )
  STORE BY PRIMARY KEY;
```

NonStop SQL/MX will place the primary partition on $VOL1, the second partition on $VOL2, and the third partition on $VOL3.

- This example creates a table stored by primary key. These defaults are in effect: `POS_LOCATIONS` is set to `$VOL1, $VOL2, $VOL3` and `POS_NUM_OF_PARTNS` is set to 5.

```
CREATE TABLE SALES.ODETAIL
  ( ordernum     NUMERIC (6) UNSIGNED  NO DEFAULT  NOT NULL,
    partnum      NUMERIC (4) UNSIGNED  NO DEFAULT  NOT NULL,
    unit_price   NUMERIC (8,2)         NO DEFAULT  NOT NULL,
    qty_ordered  NUMERIC (5) UNSIGNED  NO DEFAULT  NOT NULL,
    PRIMARY KEY (ordernum, partnum) NOT DROPPABLE )
  STORE BY PRIMARY KEY;
```

NonStop SQL/MX will place the primary partition on $VOL1, the second partition on $VOL2, the third partition on $VOL3, the fourth on $VOL1, and the fifth on $VOL2.

- This example creates a table stored by primary key. This statement includes a LOCATION clause. These defaults are in effect: `POS_LOCATIONS` is set to `$VOL1, $VOL2, $VOL3` and `POS_NUM_OF_PARTNS` is set to 4.

```
CREATE TABLE SALES.ODETAIL
  ( ordernum     NUMERIC (6) UNSIGNED  NO DEFAULT  NOT NULL,
    partnum      NUMERIC (4) UNSIGNED  NO DEFAULT  NOT NULL,
    unit_price   NUMERIC (8,2)         NO DEFAULT  NOT NULL,
    qty_ordered  NUMERIC (5) UNSIGNED  NO DEFAULT  NOT NULL,
    PRIMARY KEY (ordernum, partnum) NOT DROPPABLE )
  LOCATION \NODE3.$DATA1
  STORE BY PRIMARY KEY;
```

NonStop SQL/MX will place the primary partition on $DATA1, the second partition on $VOL1, the third partition on $VOL2, and the fourth partition on $VOL3.

IDENTITY column examples

- This example shows how to create an IDENTITY column for a simple table. In this example, the column `surrogate_key` is defined as the IDENTITY column and is the primary key of the table `t_id_S`.

```
CREATE TABLE t_id_S (surrogate_key LARGEINT GENERATED BY
DEFAULT AS IDENTITY NOT NULL,
         name CHAR (5) NOT NULL,
         primary key(surrogate_key)
         )
   HASH PARTITION BY(surrogate_key);
```

- This example shows the IDENTITY column `id_key` as part of the clustering key (STORE BY clause):

```
CREATE TABLE t_id (id_key LARGEINT GENERATED BY
                    DEFAULT AS IDENTITY NOT NULL,
                name CHAR (256) NOT NULL,
                order_number INT UNSIGNED NOT NULL
         )
      STORE BY (id_key, order_number);
```

- This example shows the IDENTITY column `id_key` as the partitioning key:

Note. In SQL/MX, the partitioning key must be a subset of the clustering key. In the case of a table with a single column clustering key, the partitioning key must be the same as the clustering key.

```
CREATE TABLE t_id (id_key LARGEINT GENERATED BY
                   DEFAULT AS IDENTITY NOT NULL,
                   name CHAR (256) NOT NULL,
                   order_number INT UNSIGNED NOT NULL
                   )
       STORE BY (id_key, order_number)
       HASH PARTITION BY(id_key);
```

- This example shows that the values for the IDENTITY column `Id_col` will always be generated by the system. MINVALUE, MAXVALUE, and NO CYCLE will take default values because they are not specified:

```
CREATE TABLE tbl1 (
       Id_col INTEGER UNSIGNED GENERATED ALWAYS AS IDENTITY
       (
       START WITH 1
       INCREMENT BY 2) NOT NULL,
       Col2 INTEGER NOT NULL, PRIMARY KEY(Id_col)
       );
INSERT INTO tbl1 values (DEFAULT, 10), (DEFAULT, 20), (DEFAULT,
30);
```

will result in the following rows inserted into table `tbl1`; (1,10), (3,20), (5,30).

```
INSERT INTO tbl1 values (15, 10);
```

will result in a error 3428 indicating that you cannot specify a value for the IDENTITY column defined as GENERATED ALWAYS.

```
*** ERROR[3428] IDENTITY column ID_COL defined as GENERATED ALWAYS
cannot accept  values specified by the user.
```

- This example fails with an error indicating that the start value must be less than the MAXVALUE and greater than the MINVALUE:

```
CREATE TABLE tbl1 (
     Id_col INTEGER UNSIGNED GENERATED BY DEFAULT AS IDENTITY
     (START WITH 100 INCREMENT BY 2 MAXVALUE 10 MINVALUE 50)
     NOT NULL,
     Col2 INTEGER NOT NULL, PRIMARY KEY(Id_col)
     );
```

```
*** ERROR[1570] The MAXVALUE for the sequence generator must be
greater than the MINVALUE for IDENTITY column ID_COL.
```

- In this example, none of the sequence generator options are specified; the default values for all options are used:

```
   start value: 0 (zero)
   increment: 1
   min value: 0 (zero)
   max value: 4294967295
   NO CYCLE
```

```
CREATE TABLE tbl1 (
  Id_col INTEGER UNSIGNED GENERATED BY DEFAULT AS IDENTITY   NOT
NULL,
  Col2 INTEGER NOT NULL, PRIMARY KEY(Id_col)
   );

   showddl tbl1;

   CREATE TABLE CAT.SCH.TBL1
    (ID_COL INT UNSIGNED GENERATED BY DEFAULT AS IDENTITY (START
WITH 0 INCREMENT BY 1 MAXVALUE 4294967295 MINVALUE 0 NO CYCLE)
     LOCATION \DMR15.$SYSTEM.ZSDWDPR4.WPBXMX00
     -- NOT NULL NOT DROPPABLE
   , COL2 INT NO DEFAULT -- NOT NULL NOT DROPPABLE
   , CONSTRAINT CAT.SCH.TBL1_697159451_3816 PRIMARY KEY (  ID_COL
ASC) NOT DROPPABLE
   , CONSTRAINT CAT.SCH.TBL1_232649451_3816 CHECK
(CAT.SCH.TBL1.ID_COL IS NOT NULL AND CAT.SCH.TBL1.COL2 IS NOT
NULL) NOT DROPPABLE
   )
   LOCATION \DMR15.$SYSTEM.ZSDWDPR4.S5RXMX00
   NAME DMR15_SYSTEM_ZSDWDPR4_S5RXMX00
   ATTRIBUTES BLOCKSIZE 4096
   STORE BY (ID_COL ASC)
   ;
```

- This example shows that the IDENTITY column options can be specified in any order:

```
CREATE TABLE tbl1 (
  Id_col INTEGER UNSIGNED GENERATED BY DEFAULT AS IDENTITY
  ( START WITH 100 MAXVALUE 1000 INCREMENT BY 2 MINVALUE 50)
  NOT NULL,
```

```
  Col2 INTEGER NOT NULL, PRIMARY KEY(Id_col)
  );
```

- This example fails with an error stating that a table can have only one IDENTITY column.

```
CREATE TABLE T (id_key LARGEINT GENERATED BY DEFAULT AS IDENTITY
NOT NULL PRIMARY KEY,
    name CHAR (256) NOT NULL,
    order_number LARGEINT GENERATED BY DEFAULT AS IDENTITY  NOT
NULL)
    HASH PARTITION BY(id_key);
```

```
*** ERROR[1511] There can only be one IDENTITY column for a table.
```

- In this example, an IDENTITY column is defined on a range partitioned table:

```
CREATE TABLE tab1(a LARGEINT GENERATED ALWAYS AS IDENTITY
  (START WITH 51 INCREMENT BY 2 MAXVALUE 55 MINVALUE 50 NO CYCLE)
NOT NULL,
  b INT UNSIGNED NOT NULL,
  c INT NOT NULL,
  d INT NOT NULL,
  e INT NOT NULL,
PRIMARY KEY(a, B, C) )
  LOCATION $data10
PARTITION BY(a,B,C)(
  ADD FIRST KEY (60, 0, 1)
    LOCATION $data12
  , ADD FIRST KEY (70, 0, 1)
    LOCATION $data13
  , ADD FIRST KEY (65, 0, 1)
    LOCATION $data14
);
```

- This example shows how to specify the location for the SG Table using the LOCATION clause:

```
CREATE TABLE tab1(a LARGEINT GENERATED ALWAYS AS IDENTITY
  (START WITH 51 INCREMENT BY 2 MAXVALUE 55 MINVALUE 50 NO CYCLE)
  LOCATION $data14,
b INT UNSIGNED NOT NULL,
c INT NOT NULL);
showddl tab1;
```

```
CREATE TABLE CAT.SCH.TAB1
  (A LARGEINT GENERATED ALWAYS AS IDENTITY
     (START WITH 51 INCREMENT BY 2 MAXVALUE 55 MINVALUE 50 NO
CYCLE)
     LOCATION \DMR15.$DATA14.ZSDWDPR4.GQ4QB600
     -- NOT NULL NOT DROPPABLE
  ,B INT UNSIGNED NO DEFAULT
      -- NOT NULL NOT DROPPABLE
  ,C INT NO DEFAULT -- NOT NULL NOT DROPPABLE
     , CONSTRAINT CAT.SCH.TAB1_156576774_5816 CHECK
(CAT.SCH.TAB1.A IS NOT NULL
      AND CAT.SCH.TAB1.B IS NOT NULL AND CAT.SCH.TAB1.C IS NOT
NULL) NOT DROPPABLE
  )
  LOCATION \DMR15.$SYSTEM.ZSDWDPR4.DDBRB600
  NAME DMR15_SYSTEM_ZSDWDPR4_DDBRB600
```

```
    ATTRIBUTES BLOCKSIZE 4096
    NO PARTITION
    ;
--- SQL operation complete.
```

CREATE TRIGGER Statement

Considerations for CREATE TRIGGER
Examples of CREATE TRIGGER

The CREATE TRIGGER statement is used to create triggers on SQL/MX tables. A trigger is a mechanism that causes the database system to perform certain actions automatically in response to specified events.

```
CREATE TRIGGER trigger-name
{BEFORE | AFTER}
   {INSERT | DELETE | UPDATE [OF (columns)]}
   ON table-name
     [REFERENCING old-new-alias-list ]
        [FOR EACH {ROW | STATEMENT}]
           [WHEN (search-condition)]
             triggered-SQL-statement;
columns is:
   column-name, columns | column-name

old-new-alias-list is:
  old-new-alias, old-new-alias-list | old-new-alias

old-new-alias  is:
   OLD [AS] correlation-name  |
   NEW [AS] correlation-name  |
   OLD [AS] table-alias
   NEW [AS] table-alias

triggered-SQL-statement is:
  searched-update-statement  |
  searched-delete-statement |
  call-statement   |
  insert-statement |
  signal-statement |
  set-new-statement

signal-statement is:
   SIGNAL SQLSTATE quoted-sqlstate (quoted-string-expr);
```

Syntax Description of CREATE TRIGGER

trigger-name

specifies the ANSI logical name of the trigger to be added, of the form:

[[*catalog-name*.]*schema-name*.]*trigger-name*

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

column-name

specifies the ANSI logical name of the column to be inserted, deleted, or updated when the trigger is activated, of the form:

[[*catalog-name*.]*schema-name*.]*column-name*

where each part of the name is a valid SQL identifier with a maximum of 128 characters.

table-name

specifies the ANSI logical name of the table this trigger is defined on, of the form:

[[*catalog-name*.]*schema-name*.]*table-name*

where each part of the name is a valid SQL identifier with a maximum of 128 characters. Triggers can be defined only on SQL/MX tables.

FOR EACH { ROW | STATEMENT }

specifies whether the trigger is based on a row or a statement. If you do not specify this clause, the default is ROW for a BEFORE trigger and STATEMENT for an AFTER trigger.

old-new-alias

is the list of correlation names or table aliases used by a trigger.

correlation-name

is the name of the old or new row acted upon by the trigger.

table-alias

is the name of the old or new table acted upon by the trigger.

search-condition

is the condition that, when true, activates this trigger. Starting with SQL/MX Release 3.2, AFTER Triggers support subqueries in the *search-condition*. Simple subqueries with joins and GROUP BY are supported.

triggered-SQL-statement

is the SQL statement to be performed when this trigger is activated.

searched-update-statement

is an update statement to be performed when the AFTER trigger is activated.

searched-delete-statement

is a delete statement to be performed when the AFTER trigger is activated.

call-statement

is a CALL statement to be performed when the AFTER trigger is activated. This support is available from SQL/MX Release 3.2.

insert-statement

is an insert statement to be performed when the AFTER trigger is activated.

signal-statement

is a statement to be sent to the SIGNAL statement.

set-new-statement

is an assignment statement that can be used as a BEFORE trigger action to assign values to transition variables representing columns in the subject table modified by the triggering action.

quoted-sqlstate

is the five-digit SQLSTATE to be passed to SIGNAL. Use the GET DIAGNOSTICS command to retrieve *quoted-string-expr* (as *message-text*) and *quoted-sqlstate*.

quoted-string-expr

is a string expression.

Considerations for CREATE TRIGGER

Triggers support up to 16 levels of recursion. Triggers have their own namespace.

SHOWDDL for a table displays the DDL CREATE text for all triggers defined on that table. The LIKE option of CREATE TABLE ignores triggers.

Triggers and Utilities

- The DUP utility does not duplicate triggers.

- By default, using the IMPORT utility to import data to a table causes trigger actions to be performed. If trigger actions are not required, use the `-d` option with the IMPORT utility, which allows triggers to be disabled for the duration of the operation.
- Most of MODIFY’s partition management operations ignore triggers. However, the REUSE form of MODIFY might return errors on a table with DELETE triggers.
- Use PURGEDATA to purge the data of a table that is referenced by a trigger or that is the subject of a trigger. PURGEDATA supports an option that indicates whether DELETE triggers on the table are ignored. If they are not ignored and a DELETE trigger exists, PURGEDATA fails.
- BACKUP/RESTORE operations on tables ignore triggers.
- The mxexportddl utility handles triggers correctly.
- VERIFY operations ignore triggers.

Authorization and Availability Requirements

To create a trigger, you must own the schema where the trigger is defined and the schema where the subject table of the schema resides and you must have REFERENCES privileges on the columns used on the referenced table. Otherwise, you must be the super ID.

Trigger Types

You can create a trigger as a BEFORE or AFTER trigger. When a triggering statement occurs, the following is the order of execution:

1. BEFORE triggered statements
2. Triggering statement
3. Referential actions
4. AFTER triggered statements

Execution of a statement is considered to be complete only when all cascaded triggers are complete. When multiple triggers are activated by the same event (that is, a conflict set), the next trigger from the original conflict set is considered only after the execution of cascaded triggers of a specific trigger is complete (depth-first execution). Within a conflict set, the order of execution is by timestamp of creation of the corresponding trigger. Older triggers are executed first.

Statement triggers and row triggers can participate in the same conflict set and can cascade each other. Therefore, they can appear intertwined.

Triggers use transition tables or transition variables to access old and new states of the table or row. Statement triggers use transition tables. Row triggers use transition variables. The following table summarizes the transition variables that different trigger types can use. “new row” refers to the transition variable and “new table” refers to the transition table:

Triggering Event and Activation Time	Row Trigger Can Use:	Statement Trigger Can Use:
BEFORE INSERT	new row	Invalid
BEFORE UPDATE	old row, new row	Invalid
BEFORE DELETE	old row	Invalid
AFTER INSERT	new row	new table
AFTER UPDATE	old row, new row	old table, new table
AFTER DELETE	old row	old table

BEFORE Triggers

BEFORE triggers are used for one of these purposes:

- To generate an appropriate signal when an insert, update, or delete operation is applied and a certain condition is satisfied (using the SIGNAL statement as an action).
- To format input data before inserting or updating the subject table that caused the trigger to be activated (using the SET statement as an action).

BEFORE-type trigger operations are exercised as tentative executions. The triggering statement is executed but assigns values to the NEW ROW transition variables rather than to the subject table. That table appears not to be affected by the tentative execution. When it is accessed by the trigger action, it shows values in place before the action of the trigger. Because BEFORE-triggers can only be row triggers, they use transition variables to access old and new states of the row.

Before-type triggers do not modify tables. However, by using a SET statement, they can assign new values only to the NEW ROW transition variables. As a result, a BEFORE-type trigger can override the effect of the original triggering statement.

The unique features of BEFORE-type triggers are:

- The triggering statement executes only after the trigger is executed.
- Only row granularity is allowed.
- Only the NEW ROW transition variable can be modified.
- BEFORE-type triggers cannot be cascading.

One of the key differences between BEFORE- and AFTER-type triggers is their relationship to constraints. A BEFORE-type trigger can prevent the violation of a constraint, whereas an AFTER-type trigger cannot, because it is executed after the constraints are checked. BEFORE-type triggers are used to condition input data, while AFTER-type triggers encode actual application logic.

Restrictions on Triggers

- The trigger feature does not allow the use of:
  - Publish/Subscribe's embedded update and embedded delete statements as triggering actions or events.
  - INSERTs, UPDATEs, and DELETEs found in compound statements delimited by BEGIN … END as triggering events.
  - Compound statements delimited by BEGIN … END as part of a triggered action.
  - Positioned deletes and updates as triggered statements.
- Do not use triggers on SQL/MX user metadata (UMD) tables, system metadata, and NonStop MXCS metadata tables.
- You cannot define triggers on SQL/MP objects. SQL/MP objects cannot be referenced in a trigger.
- Triggers are not allowed on SQL/MP aliases.
- All types of subqueries are not supported in `search-condition` for AFTER triggers:
  - Nested subqueries, queries containing UNION construct and compound statements are not allowed.
  - The transition table cannot be referenced in the WHEN clause of an AFTER trigger statement.
  - INSERT, DELETE, and UPDATE queries are not allowed in the WHEN clause of an AFTER trigger statement.

Recompilation and Triggers

User applications that change (INSERT, UPDATE, or DELETE) information in a table are automatically recompiled when a trigger with a matching event is added or dropped. User applications that use a SELECT on the subject table do not require recompilation. User applications do not require an SQL compilation when a trigger is changed from DISABLED to ENABLED, or from ENABLED to DISABLED, using the ALTER TRIGGER statement. User applications require SQL recompilations only when triggers are added or dropped. No source code changes or language compilations are required.

Triggers and Primary Keys

Suppose you create this table:

```
CREATE TABLE t1( c1 varchar(2040) NOT NULL,
                  c2 INT,
                  c3 INT,
                  c4 CHAR(3),
                  c5 CHAR(3),
                  PRIMARY KEY (c1)
                 );
 CREATE TABLE t2 (c1 CHAR(3), c2 CHAR(3));
```

When you try to create a trigger on this table using these commands, you receive errors:

```
CREATE TRIGGER trg1
        AFTER INSERT ON t1
        REFERENCING NEW AS newrow
        FOR EACH ROW
          WHEN (newrow.c2 > newrow.c3)
          INSERT INTO t2 VALUES (newrow.c4, newrow.c5);
*** ERROR[1085] The calculated key length is greater than
2048 bytes.

*** ERROR[11041] Temporary table could not be created!  Check
default partitions.
```

Trigger temporary table is a table associated with the subject table of the trigger and is used to store intermediate results during trigger execution. In the example, error is returned because of the way trigger temporary tables are created. The temporary table is created with two more columns than its corresponding subject table. The combined length of the additional columns is 16 bytes. The two added columns, along with the subject table's primary key, form the primary key of the temporary table. This primary key is too long.

Note. If you want to create triggers on a table, its primary key length cannot exceed 2032 bytes. A table that does not include triggers can have a primary key of 2048 bytes.

If you update the length of column c1 of table t1 from varchar (2040) to a varchar of 2032 or less bytes (for example, varchar (2000)), the CREATE TRIGGER statement completes successfully.

Rowsets

SQL/MX rowsets are allowed in UPDATE and DELETE statements that are trigger events.

UPDATE and DELETE statements that use rowset arrays perform multiple executions of UPDATE or DELETE statements. UPDATE and DELETE statement triggers behave as a sequence of statement triggers that are triggered once for each value in the array of values in the rowset.

This behavior is different from a row trigger because each value in the rowset might match multiple rows in the subject table. Therefore, multiple rows might be affected (updated or deleted) before the action of the trigger is executed.

Contrast this behavior with row triggers where the trigger action is executed once for each affected row.

For INSERT statement that use rowsets, an INSERT statement trigger is triggered once for the entire rowset.

Examples of CREATE TRIGGER

Before and After Triggers

Suppose that you have a database to record patients' vital signs and drugs prescribed for them. The database consists of these tables:

- `vital_signs`, which records vital signs at each visit
- `prescription`, which records prescriptions written for each patient
- `generic_drugs`, which lists generic drug equivalents for brand-name drugs

The prescription table is created like this:

```
CREATE TABLE prescription
( id                  INTEGER        NOT NULL
                                     NOT DROPPABLE,
 pat_id               INTEGER        NOT NULL,
 issuing_phys_id      INTEGER        NOT NULL,
 date_prescribed      DATE           DEFAULT NULL,
 drug                 VARCHAR(80)    DEFAULT NULL,
 record_id            INTEGER        NOT NULL,
 dosage               VARCHAR(30)    NOT NULL,
 frequency            VARCHAR(30)    DEFAULT NULL,
 refills_remaining    INTEGER        DEFAULT NULL,
 instructions         VARCHAR(255)   DEFAULT NULL,
 primary key (id))
STORE BY PRIMARY KEY
ATTRIBUTES EXTENT (1024,1024) MAXEXTENTS 700
LOCATION $D00001.ZSDDEMO1.PRSCR000;
```

You can create a BEFORE trigger on `prescription` so that when a prescription is entered, if the prescribed drug is found in `generic_drugs`, a generic drug is substituted for the brand-name drug, and the instructions for the drugs are updated:

```
CREATE TRIGGER alternate_drug
  BEFORE INSERT ON prescription
    REFERENCING NEW AS newdrug
    FOR EACH ROW
    WHEN (upshift(newdrug.drug) IN
        (SELECT upshift(generic_drugs.drug) FROM generic_drugs))
    SET newdrug.drug = (SELECT
        upshift(generic_drugs.alternate_drug)
```

```
                  FROM generic_drugs
                  WHERE upshift(newdrug.drug) =
        upshift(generic_drugs.drug))
       ,newdrug.instructions = newdrug.instructions ||
      ' Prescribed drug changes to alternative drug.';
```

You can create an AFTER trigger on `vital_signs` so that when that table is updated, NonStop SQL/MX checks the patient's weight and height. Based on their values, this trigger might add a record to `prescription` to create a new prescription for a weight-loss drug with instructions that indicate that this is a free sample:

```
CREATE TRIGGER free_sample
  AFTER INSERT ON vital_signs
    REFERENCING NEW AS sample
    FOR EACH ROW
    WHEN (sample.weight > 299 and sample.height < 69)
    INSERT INTO prescription
     (id, pat_id, issuing_phys_id, record_id, date_prescribed,
drug, dosage,
      frequency, refills_remaining, instructions)
    VALUES
    ((SELECT sequence + 1 from prescription_seq),
     (SELECT pat_id    FROM record WHERE sample.id =
record.vital_id),
     (SELECT phys_id   FROM record WHERE sample.id =
record.vital_id),
     (SELECT record.id FROM record WHERE sample.id =
record.vital_id),
     CURRENT_DATE, 'POUND OFF', '200 mg', '1 pill 1 hour before
each meal', 0, 'Free sample no refills'
    );
```

- This example

Rowsets and Triggers

Suppose that you have a table with this rowset definition:

```
Rowset[10] short ArrayA;
```

This embedded DML statement inserts ten rows into table tab1.

```
EXEC SQL insert into cat.sch.tab1 values (:ArrayA);
```

If trigger `trg1` is defined as an insert statement trigger on `tab1`,and `trg2` is defined as an insert row trigger on `tab1`, when the DML statement is executed, the two triggers are fired. The action of `trg1` executes once for the entire statement, while `trg2` executes ten times, once for each element in the rowset.

Stored Procedures and Triggers

Starting with SQL/MX Release 3.2, the trigger statements support stored procedures. The considerations are:

- IN type of procedure parameters are supported.

- SPJ with resultset is not supported.
- AFTER triggers are supported.
- You must ensure that there is no recursion between SPJ and trigger tables as it can cause undefined behavior.

The following example creates a trigger that executes an stored procedure, named LOWERPRICE defined on page 2-93, when the QTY_ON_HAND column of the PARTLOC table is updated and exceeds 500 parts. For definition of tables, see Appendix D, Sample Database.

```
CREATE TRIGGER sales.setsalesprice
AFTER UPDATE OF qty_on_hand
ON invent.partloc
FOR EACH STATEMENT
REFERENCING NEW as newqty
WHEN ( SUM(newqty.qty_on_hand) > 500 )
CALL sales.lowerprice();
```

CREATE VIEW Statement

Considerations for CREATE VIEW
Examples of CREATE VIEW

The CREATE VIEW statement creates an SQL/MX view. See Views on page 6-115.

```
CREATE VIEW view
   [(column-name [heading] [,column-name [heading]]...)]
   [ {ENABLE | DISABLE} SIMILARITY CHECK ]
   [LOCATION [\node.]$volume[.subvolume.filename]]
   AS query-expr
   [WITH [CASCADED] CHECK OPTION]

heading is:
   HEADING 'heading-string' | NO HEADING

query-expr is:
   non-join-query-expr | joined-table

non-join-query-expr is:
   non-join-query-primary | query-expr UNION [ALL] query-term

query-term is:
   non-join-query-primary | joined-table

non-join-query-primary is:
   simple-table | (non-join-query-expr)

joined-table is:
   table-ref [NATURAL] [join-type] JOIN table-ref [join-spec]

join-type is:
   INNER | LEFT [OUTER] | RIGHT [OUTER]

join-spec is:
   ON condition

simple-table is:
     VALUES (row-value-const) [,(row-value-const)]...
   | TABLE table
   | SELECT [ALL | DISTINCT] select-list
        FROM table-ref [,table-ref]...
        [WHERE search-condition]
        [SAMPLE sampling-method]
        [TRANSPOSE transpose-set [transpose-set]...
           [KEY BY key-colname]]...
        [SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
           [,colname [ASC[ENDING] | DESC[ENDING]]]...]
       [GROUP BY {colname | colnum} [,{colname |
colnum}]...]
        [HAVING search-condition]

row-value-const is:
   row-subquery | expression [,expression]...
```

Syntax Description of CREATE VIEW

view

specifies the ANSI logical name for the view of the form:

`[[catalog-name.]schema-name.]view`

where each part of the name is a valid SQL identifier with a maximum of 128 characters. *view* must be unique among table, view, SQL/MP alias, and procedure names in the schema. For more information, see Identifiers on page 6-56.

`(column-name [heading] [,column-name [heading]]...)`

specifies names for the columns of the view and, optionally, headings for the columns. Column names in the list must match one-for-one with columns in the table specified by *query-expr*.

If you omit this clause, columns in the view have the same names as the corresponding columns in *query-expr*. You must specify this clause if any two columns in the table specified by *query-expr* have the same name or if any column of that table does not have a name. For example, in the query expression "SELECT MAX(salary), AVG(salary) AS average_salary FROM employee" the first column does not have a name.

column-name

specifies the name for a column in the view. *column-name* is an SQL identifier. *column-name* must be unique among column names in the view and cannot be a reserved word. It can contain a reserved word if it is delimited.

If you do not specify this clause, columns in the view have the same names as the columns in the select list of *query-expr*.

No two columns of the view can have the same name; if a view refers to more than one table and the select list refers to columns from different tables with the same name, you must specify new names for columns that would otherwise have duplicate names.

`HEADING 'heading-string' | NO HEADING`

specifies a string *heading-string* of 0 to 128 characters to use as a heading for the column if it is displayed by using a SELECT statement in MXCI. The *heading-string* can contain characters only from the ISO88591 character set. The default heading is the column name. If you specify a heading that is identical to the column name, INVOKE and SHOWDDL do not display that heading.

If you specify NO HEADING or HEADING '', NonStop SQL/MX stores this as HEADING '', and the column name is displayed as the heading in a SELECT statement. The behavior for HEADING '' is different from that of NonStop

SQL/MP, which does not display anything for a heading in a SELECT statement if the heading is specified as HEADING ‘’.

The HEADING clause is an SQL/MX extension.

`ENABLE SIMILARITY CHECK`

enables Similarity Check for the view.

`DISABLE SIMILARITY CHECK`

disables Similarity Check for the view.

`LOCATION [\`*`node`*`.]`*`$volume`*`[.`*`subvolume`*`.`*`filename`*`]`

specifies a node and volume for the label of the view.

`node`

is the name of a node on the Expand network.

For Guardian files representing a table or index partition or a view label, *`node`* can be any node from which the object's catalog is visible.

`volume`

is the name of an audited, non-SMF DAM volume on the specified node (or the Guardian volume named in the =_DEFAULTS define if none is specified).

If you do not specify a LOCATION clause and your system does not have a value for the DDL_DEFAULT_LOCATIONS default (either in your environment or at the system level) and your environment does not have a =_DEFAULTS value, the CREATE statement will fail with an error.

`subvolume`

is the designated schema subvolume for the schema in which the index is being created.

Follow these guidelines when using SQL/MX subvolume names:

- The name must begin with the letters *`ZSD`*, followed by a letter, not a digit (for example, ZSDa, not ZSD2).
- The name must be exactly eight characters long.

`file-name`

is an optional Guardian file name. *`file-name`* names must be eight characters long and must end with the digits “00” (zero zero).

When you specify the *`subvolume`*, the *`file-name`* must be specified with it. The *`subvolume`* and *`file-name`* are optional.

Any Guardian file name you specify must match the designated schema subvolume name for the schema in which the object is being created. Otherwise, NonStop SQL/MX returns an error.

`AS` *`query-expr`*

specifies the columns for the view and sets the selection criteria that determines the rows that make up the view. This *`query-expr`* cannot contain non-ISO88591 string literals. For the syntax description of *`query-expr`*, see SELECT Statement on page 2-330.

`WITH [CASCADED] CHECK OPTION`

specifies that no row can be inserted or updated in the database through the view unless the row satisfies the view definition—that is, the search condition in the WHERE clause of the query expression must evaluate to TRUE for any row that is inserted or updated.

If you omit this option, a newly inserted row or an updated row need not satisfy the view definition, which means that such a row can be inserted or updated in the table but does not appear in the view. This check is performed each time a row is inserted or updated.

WITH CHECK OPTION does not affect the query expression; rows must always satisfy the view definition. CASCADED is an optional keyword; WITH CHECK OPTION has the same effect.

Considerations for CREATE VIEW

You cannot create an SQL/MX view that references an SQL/MP table or an SQL/MP alias.

VIEW SIMILARITY CHECK

Starting with SQL/MX Release 3.2, the CREATE VIEW statement provides optional syntax to enable or disable Similarity Check for a view. By specifying ENABLE

SIMILARITY CHECK, you can enable Similarity Check for a view. Similarly, by specifying DISABLE SIMILARITY CHECK, you can disable Similarity Check for a view.

Note. The CQD, DDL_VIEW_SIMILARITY_CHECK controls the Similarity Checks for views at the system level. The ENABLE or DISABLE SIMILARITY CHECK clauses in the CREATE VIEW and ALTER VIEW statements take precedence over the CQD setting. The default setting for this CQD is ENABLE.

The following lists the CQD settings and behavior:

- ENABLE/SYSTEM - If the view definition permits, Similarity Check is enabled, else Similarity Check is disabled.
- ON - If the view definition permits, Similarity Check is enabled, otherwise the view is not created or altered, and an error is returned.
- DISABLE/OFF - Similarity Check is disabled.

Restrictions for Similarity Check

- Nested views and views with a VALUES clause are not supported.
- View Similarity Check is not supported for SQL/MP views.

Reserved View Names

View names prefixed by the name of a UMD table are reserved. You cannot create views with such names. For example, you cannot create a view named HISTOGRAMS_MYVIEW.

Effect of Adding a Column on View Definitions

The addition of a column to a table has no effect on any existing view definitions or conditions included in constraint definitions. Any implicit column references specified by SELECT * in view or constraint definitions are replaced by explicit column references when the definition clauses are originally evaluated.

Authorization and Availability Requirements

To create a view, you must own the schema and have SELECT privileges for the objects underlying the view.

When you create a view on a single table, the owner of the view is automatically given all privileges WITH GRANT OPTION on the view. However, when you create a view that spans multiple tables, the owner of the view is given only SELECT privileges WITH GRANT OPTION. If you try to grant privileges to another user on the view other than SELECT you will receive a warning that you lack the grant option on that privilege.

Updatable and Non-Updatable Views

Single table views can be updatable. Multi-table views cannot be updatable.

To define an updatable view, a query expression must also meet these requirements:

- It cannot contain a JOIN, UNION, or EXCEPT clause.
- It cannot contain a GROUP BY or HAVING clause.
- It cannot directly contain the keyword DISTINCT.
- The FROM clause must refer to exactly one table or one updatable view.
- It cannot contain a WHERE clause that contains a subquery.
- The select list cannot include expressions or functions or duplicate column names.

An updatable view is *insertable* if the column list does not include a SYSKEY from the underlying base table.

Examples of CREATE VIEW

- This example creates a view on a single table without a view column list:

```
CREATE VIEW SALES.MYVIEW1 AS
  SELECT ordernum, qty_ordered FROM SALES.ODETAIL;
```

- This example creates a view with a column list:

```
CREATE VIEW SALES.MYVIEW2
  (v_ordernum, t_partnum) AS
  SELECT v.ordernum, t.partnum
    FROM SALES.MYVIEW1 v, SALES.ODETAIL t;
```

- This example creates a view WITH CHECK OPTION:

```
CREATE VIEW SALES.MYVIEW3
  (ordernum HEADING 'Number of Order') AS
  SELECT ordernum FROM SALES.ODETAIL
    WHERE partnum < 1000 WITH CHECK OPTION;
```

- This example creates a view from two tables by using an INNER JOIN:

```
CREATE VIEW MYVIEW4
  (v_ordernum, v_partnum) AS
  SELECT od.ordernum, p.partnum
    FROM SALES.ODETAIL OD INNER JOIN SALES.PARTS P
    ON od.partnum = p.partnum;
```

- This example enables Similarity Check for a view:

```
CREATE VIEW MYVIEW5
  (v_ordernum, v_partnum) ENABLE SIMILARITY CHECK AS
  SELECT od.ordernum, p.partnum
    FROM SALES.ODETAIL OD INNER JOIN SALES.PARTS P
    ON od.partnum = p.partnum;
```

- This example disables Similarity Check for a view:

```
CREATE VIEW MYVIEW5
  (v_ordernum, v_partnum) DISABLE SIMILARITY CHECK AS
```

```
SELECT od.ordernum, p.partnum
  FROM SALES.ODETAIL OD INNER JOIN SALES.PARTS P
  ON od.partnum = p.partnum;
```

DELETE Statement

Considerations for DELETE
MXCI Examples of DELETE
C Examples of DELETE
COBOL Examples of DELETE
Publish/Subscribe Examples of DELETE

The DELETE statement is a DML statement that deletes a row or rows from a table or an updatable view. Deleting rows from a view deletes the rows from the table on which the view is based. DELETE does not remove a table or view, even if you delete the last row in the table or view.

Starting with SQL/MX Release 3.2, self-referencing deletes are supported. With this support, you can select the rows for delete from the target table in a subquery.

The three forms of the DELETE statement are:

- Searched DELETE— Delete rows (the selection of which depends on a search condition)
- **Embed** Positioned DELETE— Delete a single row that is determined by the cursor position. ■
- MULTI COMMIT DELETE (MCD)— Delete a large number of records in a series of smaller independent transactions.

For the searched DELETE form, if there is no WHERE clause, all rows are deleted from the table or view.

Use the positioned form of DELETE only in embedded SQL programs. Use the searched form in MXCI or embedded SQL programs.

Searched DELETE is:

Embed
```
[ ROWSET FOR INPUT SIZE rowset-size-in ] ■

DELETE [multi-commit-option] FROM table
```
Pub/Sub
```
 | STREAM (table) [AFTER LAST ROW] ■
```
Pub/Sub
```
   [SET ON ROLLBACK set-roll-clause [,set-roll-clause]...] ■
```
Embed
```
[WHERE search-condition | rowset-search-condition] ■

   [[FOR] access-option ACCESS]
```
Pub/Sub
```
set-roll-clause is:
   column-name = expression | ■

access-option is:
   READ COMMITTED
 | SERIALIZABLE
 | REPEATABLE READ
 | SKIP CONFLICT
```

Embed

Positioned DELETE is:

```
DELETE FROM table
```
C/COBOL
```
   WHERE CURRENT OF {cursor-name | ext-cursor-name} ■
■
```

MULTI COMMIT DELETE is:

```
multi-commit-option is:
   WITH MULTI COMMIT [ granularity ]

granularity is:
   EVERY num ROWS
```

Embed

`ROWSET FOR INPUT SIZE` *`rowset-size-in`*

Allowed only if you specify *`rowset-search-condition`* in the WHERE clause. *`rowset-size-in`* restricts the size of the input rowset to the specified size. If *`rowset-size-in`* is different from the allocated size for the rowset, NonStop SQL/MX uses the smaller of the two sizes and ignores the remaining entries in the larger rowset.

`rowset-size-in` must be an integer literal (exact numeric literal, dynamic parameter, or a host variable) whose type is unsigned short, signed short, unsigned long, or signed long in C and their corresponding equivalents in COBOL.

If you do not specify `rowset-size-in`, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section of the embedded SQL program. ■

`table`

names the user table or view from which to delete rows. *`table`* must be either a base table or an updatable view. To refer to a table or view, use one of these name types:

- Guardian physical name
- ANSI logical name
- DEFINE name

See Database Object Names on page 6-13.

The file organization of the table or base table must be key-sequenced. You cannot use DELETE to delete rows from an SQL/MP entry-sequenced table.

Pub/Sub `STREAM (`*`table`*`)`

deletes a continuous data stream from the specified table. You cannot specify STREAM access for the DELETE statement if it is not embedded as a table reference in a SELECT statement. See SELECT Statement on page 2-330.

`[AFTER LAST ROW]`

causes the stream to skip all existing rows in the table and delete only rows that are published after the stream's cursor is opened. ■

Pub/Sub `SET ON ROLLBACK` *`set-roll-clause`* `[,`*`set-roll-clause`*`]...`

causes one or more columns to be updated when the execution of the DELETE statement causes its containing transaction to be rolled back.

`set-roll-clause`

sets the specified column to a particular value. For each *`set-roll-clause`*, the value of the specified target *`column-name`* is replaced by the value of the update source *`expression`*. The data type of each target column must be compatible with the data type of its source value.

`column-name`

names a column in *`table`* to update. You cannot qualify or repeat a column name. You cannot update the value of a column that is part of the primary key.

`expression`

is an SQL value expression that specifies a value for the column. The *`expression`* cannot contain an aggregate function defined on a column. The data type of *`expression`* must be compatible with the data type of *column-name*. A scalar subquery in *expression* cannot refer to the table being updated.

If *expression* refers to columns being updated, NonStop SQL/MX uses the original values to evaluate the expression and determine the new value.

See Expressions on page 6-41.

`WHERE` *`search-condition`*

specifies a search condition that selects rows to delete. Within the search condition, any columns being compared are columns in the table or view being deleted from. See Search Condition on page 6-108.

If you do not specify a search condition, all rows in the table or view are deleted. You can also delete all the rows from a table or a partition of a table by using the PURGEDATA utility.

Embed `WHERE` *`rowset-search-condition`*

specifies an array of search conditions that selects rows to delete. The search conditions are applied successively and rows selected by each condition are deleted before the next search condition is applied. See Rowset Search Condition on page 6-110. ■

`[FOR]` *`access-option`* `ACCESS`

specifies the access option required for data used in the evaluation of the search condition. See Data Consistency and Access Options on page 1-8.

`READ COMMITTED`

specifies that any data used in the evaluation of the search condition must come from committed rows.

`SERIALIZABLE | REPEATABLE READ`

specifies that the DELETE statement and any concurrent process (accessing the same data) execute as if the statement and the other process had run serially rather than concurrently.

`SKIP CONFLICT`

enables transactions to skip rows locked in a conflicting mode by another transaction. The rows under consideration are the result of evaluating the search condition for the DELETE statement. SKIP CONFLICT cannot be used in a SET TRANSACTION statement.

The default access option is the isolation level of the containing transaction, which is determined according to the rules specified in Isolation Level on page 10-56.

C/COBOL

`WHERE CURRENT OF {`*`cursor-name`*` | `*`ext-cursor-name`*`}`

specifies the name of a cursor (or extended cursor) positioned at the row to delete. If you specify *`cursor-name`* for an audited table or view, the DELETE must execute within a transaction that also includes the FETCH for the row. For more information about cursor names and extended cursor names, see DECLARE CURSOR Declaration on page 3-22 and ALLOCATE CURSOR Statement on page 3-3. ■

For more information on searched and positioned DELETE statements in embedded SQL programs, see the *SQL/MX Programming Manual for C and COBOL*.

`EVERY `*`num`*` ROWS`

specifies the number of rows to be deleted for each independent transaction for a multi commit delete operation. *`num`* must be an unsigned integer greater than zero. The default value for *`num`* is 500.

Considerations for DELETE

In a searched DELETE, rows are deleted in sequence. If an error occurs and you are not using DP2's Savepoint feature, NonStop SQL/MX returns an error message and stops deleting from the table. NonStop SQL/MX automatically rolls back the transaction to undo the deleted data from the audited table.

If the default INSERT_VSBB is set to USER, NonStop SQL/MX does not use statement atomicity. Unless you are deleting only a few records, you should not disable INSERT_VSBB to use statement atomicity, because performance is affected. Perform UPDATE STATISTICS on the tables so that row estimates are correct.

To see what rollback mode NonStop SQL/MX is choosing, you can prepare the query, and then use the EXPLAIN statement:

```
explain options 'f' my_query;
```

Token “x” means that the transaction will be rolled back. Token “s” means that NonStop SQL/MX will choose DP2 savepoints. See EXPLAIN Statement on page 2-208 for details. For details about these defaults, see INSERT_VSBB on page 10-75 and UPD_SAVEPOINT_ON_ERROR on page 10-78.

Multi Commit Delete

DELETE operations can be long running on a very large data set (the number of rows affected in a single table in a single transaction). This causes the locks to escalate to file locks resulting in a loss of concurrency.

The multi commit delete feature executes these DELETE statements in multiple statements. Each of these multiple statements is executed in a separate transaction, thus avoiding lock escalation. These multiple transactions are independent transactions from TMF. Therefore, any point-in-time recovery by TMF is done individually for each of these multiple transactions and not for the multi commit delete as a whole.

Note. The *multi-commit-option* must only be used for large tables. Using this option on small tables incurs unnecessary overhead from starting and committing multiple transactions.

This option is best used to delete unwanted data (for example, records older than 90 days) from the database. Using this option for any other purpose can have point-in-time recovery ramifications since these multiple transactions are independent transactions from TMF.

Considerations

- On successful completion, the multi commit delete statement reports the total number of rows deleted from all partitions of the table. If there is a failure, it reports the total number of rows deleted to the point of the failure from all partitions along with the diagnostic information.
- Every multi commit delete statement is treated as a new statement. If there is a failure, you can fix the error and reissue the statement. The operation scans the processed rows again.
- Multi commit delete works only when the AUTOCOMMIT mode is ON. It does not work inside an explicit transaction (started with a BEGIN WORK).

Restrictions

The multi commit delete feature has the following restrictions:

- It does not support views.
- It does not support stream access.
- It cannot be part of an embedded DELETE statement.
- A host variable cannot be used to specify the granularity of each child transaction.
- A host variable cannot be used in the WHERE clause of a multi commit delete statement.
- It does not support Java applications using the JDBC T2 driver.
- The following restrictions which apply to an embedded DELETE are also applicable to a multi commit delete:
    - An embedded DELETE cannot have a rowset search condition in the predicate of the DELETE statement; SQL error 3426 is returned.
    - An embedded DELETE cannot have a subquery in the predicate of the DELETE statement; SQL error 4139 is returned.
    - An embedded DELETE on a table, which is the subject table of a trigger, returns SQL error 11045.

Authorization Requirements

DELETE requires authority to read and write to the table or view being deleted from and authority to read tables or views specified in subqueries used in the search condition.

Transaction Initiation and Termination

The DELETE statement automatically initiates a transaction if there is no active transaction and if the statement references an audited table. Otherwise, you can explicitly initiate a transaction with the BEGIN WORK statement. When a transaction is started, the SQL statements execute within that transaction until a COMMIT or ROLLBACK is encountered or an error occurs.

Isolation Levels of Transactions and Access Options of Statements

The isolation level of an SQL transaction defines the degree to which the operations on data within that transaction are affected by operations of concurrent transactions. When you specify access options for the DML statements within a transaction, you override the isolation level of the containing transaction. Each statement then executes with its individual access option.

Note. NonStop SQL/MX accepts SQL/MP keywords as synonyms for READ UNCOMMITTED, STABLE, and SERIALIZABLE.

You can explicitly set the isolation level of a transaction with the SET TRANSACTION statement. See SET TRANSACTION Statement on page 2-376.

The default isolation level of a transaction is determined according to the rules specified in Isolation Level on page 10-56.

When you specify any statement level attribute, all attributes are used from the statement specification and they override session level attributes.

When you specify one or more SET TRANSACTION attributes at the statement level, all the other SET TRANSACTION settings revert to their default values for that statement instead of the current session-level attribute values. For example, if you specify the 'in share mode' option with a SQL Statement, the statement level options will be applied. Thus, all attributes are chosen at the statement level, including the isolation level.

Therefore, if any attributes are specified for a given statement, all other SET TRANSACTION session-level settings that do not have the default value should also be specified.

Embed

It is important to note that the SET TRANSACTION statement might cause a dynamic recompilation of the DML statements within the next transaction. Dynamic recompilation occurs if NonStop SQL/MX detects a change in the transaction mode at run time compared with the transaction mode at the time of static SQL compilation. To avoid dynamic recompilation because of a change in the transaction mode, consider specifying access options for individual DML statements instead of using SET TRANSACTION. ■

Audited and Nonaudited Tables

SQL/MX tables can only be audited. You can run NonStop SQL/MX against nonaudited SQL/MP tables.

The TMF product works only on audited tables, so a transaction does not protect operations on nonaudited tables. Nonaudited tables follow a different locking and error handling model than audited tables. Certain situations, such as DML error occurrences or utility operations with DML operations, can lead to inconsistent data within a nonaudited table or between a nonaudited table and its indices.

To avoid problems, do not run DDL or utility operations concurrently with DML operations on nonaudited tables. When you try to delete data in a nonaudited table with an index, NonStop SQL/MX returns an error.

Pub/Sub

SET ON ROLLBACK Considerations

The SET ON ROLLBACK expression is evaluated when each row is processed during execution of the DELETE statement. The results of the evaluation are applied when and if the transaction is rolled back. This has two important implications:

- If the SET ON ROLLBACK expression generates an error (for example, a divide by zero or overflow error), the error is returned to the application when the DELETE operation executes, regardless of whether the operation is rolled back.
- If a DELETE operation is applied to a set of rows and an error is generated while executing the DELETE operation, and the transaction is rolled back, the actions of the SET ON ROLLBACK clause apply only to the rows that were processed by the DELETE operation before the error was generated. ■

Pub/Sub

SET ON ROLLBACK Restrictions

The table must be audited. The columns used in the SET ON ROLLBACK clause:

- Must be declared as NOT NULL.
- Cannot be part of a referential integrity constraint or be part of a secondary index.
- Cannot use the VARCHAR data type.
- Cannot be used in the primary key, clustering key, or partitioning key. ■

MXCI Examples of DELETE

- Remove all rows from the JOB table:

```
DELETE FROM persnl.job;

--- 10 row(s) deleted.
```

- Remove the row for TIM WALKER from the EMPLOYEE table:

```
DELETE FROM persnl.employee
WHERE first_name = 'TIM' AND last_name = 'WALKER';

--- 1 row(s) deleted.
```

- Remove from the table ORDERS any orders placed with sales representative 220 by any customer except customer number 1234:

```
DELETE FROM sales.orders
WHERE salesrep = 220 AND custnum <> 1234;

--- 2 row(s) deleted.
```

- Remove from the table PARTSUPP all suppliers who charge more than $1,600.00 for items that have part numbers in the range 6400 to 6700:

```
DELETE FROM invent.partsupp
WHERE partnum BETWEEN 6400 AND 6700
  AND partcost > 300.00 SERIALIZABLE ACCESS;

--- 3 row(s) deleted.
```

  This DELETE uses SERIALIZABLE access, which provides maximum consistency but reduces concurrency. Therefore, you should run this statement at a time when few users need concurrent access to the database.

- Remove all suppliers not in Texas from the table PARTSUPP:

```
DELETE FROM invent.partsupp
WHERE suppnum IN
  (SELECT suppnum FROM samdbcat.invent.supplier
   WHERE state <> 'TEXAS');

--- 41 row(s) deleted.
```

  This statement achieves the same result:

```
DELETE FROM invent.partsupp
WHERE suppnum NOT IN
  (SELECT suppnum FROM samdbcat.invent.supplier
   WHERE state = 'TEXAS');

--- 41 row(s) deleted.
```

-

```
drop table test1;

create table test1(col1 int not null , col2 char(3),primary
key(col1) );


insert into test1 values ( 1, '100'), (2, '200');

--- 2 row(s) inserted.

prepare s1 from delete from test1 where col1 =  (select col1
from test1 where col1 > 1);
--- SQL command prepared.

>>explain s1;

------------------------------------------------------------
----- PLAN SUMMARY

MODULE_NAME .............. DYNAMICALLY COMPILED
STATEMENT_NAME ........... S1
PLAN_ID .................. 212204692463357465
ROWS_OUT ................. 1
EST_TOTAL_COST ........... 0.03
STATEMENT ................ delete from test1
                           wherecol1=(selectcol1from
                           test1 where col1>
                           1);
------------------------------------------------------------
----- NODE LISTING
ROOT ====================================  SEQ_NO 11
ONLY CHILD 10
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0
EST_TOTAL_COST ........... 0.03
DESCRIPTION
  max_card_est ........... 1
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  statement_index ........ 0
  olt_optimization ....... not used
  affinity_value  1,309,464,585
  est_memory_per_cpu .... 56
  upd_action_on_error .... savepoint
  xn_autoabort_interval    -1
  plan_version ....... 3,200
  self_referencing_update  forced_sort
  MXCI_PROCESS ........... ON
  SHOWCONTROL_UNEXTERNALI  OFF
  BLOCK_TO_PREVENT_HALLOW  ON
```

```
  select_list ............ execution_count

>>execute s1;
--- 1 row(s) deleted.
>>select * from test1;
COL1         COL2
-----------  ----
          1  100
--- 1 row(s) selected.
```

C Examples of DELETE

- Remove the row for JOHN WALKER from the EMPLOYEE table:

```
EXEC SQL DELETE FROM PERSNL.EMPLOYEE
     WHERE FIRST_NAME = 'JOHN' AND LAST_NAME = 'WALKER'
     SERIALIZABLE ACCESS;
```

- Use a cursor and delete some of the returned rows during processing:

```
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
     SELECT EMPNUM, DEPTNUM, JOBCODE, SALARY
     FROM PERSNL.EMPLOYEE
     FOR SERIALIZABLE ACCESS
     FOR UPDATE;
...
EXEC SQL OPEN emp_cursor;
...
EXEC SQL FETCH emp_cursor
     INTO :hv_empnum, :hv_deptnum, :hv_jobcode, :hv_salary;
...                              /* Process fetched row. */
if (hv_jobcode == 1234)
   EXEC SQL DELETE FROM PERSNL.EMPLOYEE
        WHERE CURRENT OF emp_cursor;
```

COBOL Examples of DELETE

- Remove the row for JOHN WALKER from the EMPLOYEE table:

```
EXEC SQL DELETE FROM PERSNL.EMPLOYEE
     WHERE FIRST_NAME = 'JOHN' AND LAST_NAME = 'WALKER'
     SERIALIZABLE ACCESS
END-EXEC.
```

- Use a cursor and delete some of the returned rows during processing:

```
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
     SELECT EMPNUM, DEPTNUM, JOBCODE, SALARY
     FOR UPDATE ACCESS
     FROM PERSNL.EMPLOYEE
     FOR SERIALIZABLE ACCESS
     FOR UPDATE
END-EXEC.
...
EXEC SQL OPEN emp_cursor END-EXEC.
...
EXEC SQL FETCH emp_cursor
     INTO :hv-empnum, :hv-deptnum,
          :hv-jobcode, :hv-salary
END-EXEC.
     ...

* Process fetched row.

IF hv-jobcode = 1234
   EXEC SQL DELETE FROM PERSNL.EMPLOYEE
     WHERE CURRENT OF emp_cursor
   END-EXEC.
END-IF.
...
```

Publish/Subscribe Examples of DELETE

Suppose that these SQL/MP tables and index (and the metadata mappings) have been created:

```
CREATE TABLE $db.dbtab.tab1 (a INT NOT NULL, b INT, c INT);
CREATE TABLE $db.dbtab.tab2 (a INT, b INT, c INT);
CREATE INDEX $db.dbtab.itab1 ON tab1(b, c);

CREATE SQLMP ALIAS cat.sch.tab1 $db.dbtab.tab1;
CREATE SQLMP ALIAS cat.sch.tab2 $db.dbtab.tab2;
```

- This example shows the SET ON ROLLBACK clause. The SET ON ROLLBACK column must be declared as NOT NULL; it cannot be part of a secondary index.

```
SET NAMETYPE ANSI;
SET SCHEMA cat.sch;

DELETE FROM tab1
SET ON ROLLBACK a = a + 1;
```

- This example shows the SET ON ROLLBACK clause in an embedded delete of a SELECT statement:

```
SELECT * FROM
  (DELETE FROM tab1 SET ON ROLLBACK a = a + 1) tab1;
```

- This example shows SKIP CONFLICT ACCESS used with an embedded delete statement accessing a table as a stream:

  ```
  SELECT a FROM (DELETE FROM STREAM(tab1)
  WHERE a = 1 FOR SKIP CONFLICT ACCESS) as tab1;
  ```

DOWNGRADE Utility

Considerations for DOWNGRADE
Example of DOWNGRADE

DOWNGRADE is a syntax-based utility that you can run from MXCI. DOWNGRADE transforms metadata from the existing version to a specified lower version. The REPORTONLY option allows you to test if the operation can be executed without actually performing the operation.

Starting with SQL/MX Release 3.2, the DOWNGRADE utility can downgrade all metadata in a named catalog, and optionally in all catalogs related to the specified catalog.

```
DOWNGRADE ALL METADATA TO VERSION version
    [ output-spec ]

DOWNGRADE ALL METADATA IN CATALOG catalog TO VERSION version
    [ RESTRICT | CASCADE ]
    [ output-spec ]

output-spec is:
  [ log-to-spec ] [ REPORTONLY ]

log-to-spec is:
  { [ LOG TO ] OUTFILE oss-file [ CLEAR ] | LOG TO HOMETERM }
```

catalog

specifies the ANSI name of the catalog, whose metadata is to be downgraded. There is no default for *catalog*.

version

is a valid SQL/MX version number which specifies the target version of the command. *version* must be lower than the actual schema version for all schemas affected by the command.

RESTRICT

restricts the downgrade to only the metadata in the named catalog.

CASCADE

extends the downgrade to the transitive closure of catalogs that are related to the specified catalog. The default is RESTRICT.

oss-file

an OSS file name.

Note. The DOWNGRADE utility is available only on systems running J06.11 and later J-series RVUs and H06.22 and later H-series RVUs.

Considerations for DOWNGRADE

Modes of Operation for DOWNGRADE

There are two modes of operation for the DOWNGRADE utility:

- `ALL METADATA`

  This mode of operation downgrades all metadata that is visible on the system, including the metadata in the system catalog.

- `ALL METADATA IN CATALOG` *`catalog`*

  This mode of operation downgrades all metadata in the named catalog. If the `CASCADE` option is specified, it also downgrades metadata in all catalogs that are related to the named catalog. This mode of operation is available from SQL/MX Release 3.2.

Command Output for DOWNGRADE

The DOWNGRADE utility supports the following command output options:

- `REPORTONLY`

  If the `REPORTONLY` option is specified, only the initial error checking is performed; the DOWNGRADE operation is not performed. If the `LOG TO` option is also specified, the list of affected schemas is written to the output file.

- `LOG TO`

  If the `LOG TO` option is specified, the command writes a log of its progress to either the specified *oss-file* or to the home terminal. If the `CLEAR` option is used and if *oss-file* is an existing disk file, *oss-file* is cleared before logging begins. Otherwise, the output is appended to the existing contents of *oss-file*. The following is the format of the first line of log output:

```
************** Time: <time> Process: <process>  Log opened **************
```

  The format enables you to recognize a log file easily. A command is rejected if it specifies an existing non-empty *oss-file* that is not a log file.

  Log file messages correspond to the EMS event messages. Regardless of the `LOG TO` option, the DOWNGRADE utility will generate EMS events to the $0 primary collector that documents the progress of the command. For information about EMS event messages, see the *SQL/MX Messages Manual*.

Error Conditions

The following are examples of the error conditions that might occur while executing the DOWNGRADE utility:

- An affected schema has a schema version that is lower than the target version
- No schemas are affected by the operation
- An object in an affected schema has an object feature version (OFV) that is higher than the target version
- The target version is not a valid schema version
- The `RESTRICT` option is specified or no option is specified, and one or more related catalogs exist
- All the following conditions are met :
  - One or more user schemas are participating in a downgrade to version 1200
  - The system schema version is higher than 1200
  - The system schema is not involved in the downgrade
- Concurrent DOWNGRADE operations are not supported

Recovery of a Failed DOWNGRADE Utility

The RECOVER command allows for recovery of a failed DOWNGRADE command.

```
RECOVER ALL METADATA
    [ RESUME | CANCEL ]
    [ output-spec ]

RECOVER ALL METADATA IN CATALOG catalog
    [ RESUME | CANCEL ]
    [ output-spec ]
```

`catalog`

specifies the ANSI name of the catalog that is specified in the original DOWNGRADE command.

`RESUME`

enables you to continue the processing of the original command, starting at the point of interruption.

`CANCEL`

enables you to revert the changes made by the original command, thereby returning the database to its original state. The default value is CANCEL.

output-spec

corresponds to the *output-spec* for DOWNGRADE.

Note. The RECOVER command does not need CASCADE option as it automatically recovers the metadata for those catalogs that are affected by the original DOWNGRADE command.

The RECOVER command must use the same mode of operation as the original DOWNGRADE command:

- If the original DOWNGRADE uses the ALL METADATA mode of operation, then the RECOVER must also use the ALL METADATA mode of operation.
- If the original DOWNGRADE uses the ALL METADATA IN CATALOG catalog mode of operation then the RECOVER must also use the ALL METADATA IN CATALOG catalog mode of operation, and must specify the same catalog name.

In a distributed environment, the RECOVER command must be issued from the same system where the original DOWNGRADE command was run.

Error Conditions

The following are the error conditions that might occur while executing the RECOVER command:

- An involved node has an incompatible version (because the version of the node was changed between the time of the original operation and the time of recover)
- No corresponding UPGRADE or DOWNGRADE operation is recorded
- The original command is still active

Example of DOWNGRADE

This example transforms all metadata to version 1200:

```
DOWNGRADE ALL METADATA TO VERSION 1200;
```

The following is an excerpt from the output file.

```
*************** Time: <time> Process: <process>  Log opened
***************

The DOWNGRADE ALL METADATA TO VERSION 1200 has started
Schema CAT.SCH will be downgraded from version 3000 to version 1200
                     ...
Creating version 1200 definition schema for catalog CAT
Downgrading version 3000 metadata to version 1200 for affected schemas in
catalog CAT
Set schema version to 1200 for CAT.SCH
Updating file labels for affected schemas in catalog CAT
Remove CAT.DEFINITION_SCHEMA_VERSION_3000
Schema CAT.SCH has been downgraded from version 3000 to version 1200
                     ...
The DOWNGRADE ALL METADATA TO VERSION 1200 has completed
```

Note. The date-time-processid prefix of each line and the output for schemas in the system catalog are not displayed in the output file.

DROP CATALOG Statement

The DROP CATALOG statement deletes an empty SQL/MX catalog. See Catalogs on page 6-3.

DROP CATALOG is an SQL/MX extension.

```
DROP CATALOG catalog
```

Syntax Description of DROP CATALOG

catalog

is the name of the catalog to drop.

Considerations for DROP CATALOG

Drop all schemas from the catalog before you can use the DROP CATALOG statement. This statement automatically removes the SQL/MX metadata associated with the catalog.

Reserved Catalogs

Catalog names beginning with NONSTOP_SQLMX_ are reserved for system metadata. You are not allowed to drop the system metadata catalog.

Authorization and Availability Requirements

Only a catalog owner and super ID can drop a catalog. All metadata tables for the catalog must be accessible at the time DROP CATALOG executes. No user can drop a nonempty catalog, even if the catalog contains only empty schemas.

Examples of DROP CATALOG

- This example drops an empty catalog:

```
DROP CATALOG mycatalog;
```

DROP INDEX Statement

Considerations for DROP INDEX
Examples of DROP INDEX

The DROP INDEX statement deletes an SQL/MX index. See Database Object Names on page 6-13.

DROP INDEX is an SQL/MX extension.

```
DROP INDEX index [RESTRICT | CASCADE]
```

Syntax Description of DROP INDEX

index

is the ANSI logical name of the index to drop, of the form:

[[*catalog-name*.]*schema-name*.]*index*

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

If you specify RESTRICT and the index is being used by to validate a constraint, the index is not dropped.

If you specify CASCADE and the index is being used to validate a constraint, the constraint and the index are dropped.

The default is RESTRICT.

Considerations for DROP INDEX

Authorization and Availability Requirements

To drop an index, you must own the schema that contains the index or be the super ID or object owner, and have access to all partitions of the index and the underlying table.

Indexes That Support Constraints

NonStop SQL/MX uses indexes to implement some constraints. You cannot use DROP INDEX to drop an index that implements a constraint unless you use the CASCADE option. Use CASCADE to drop all constraints that use the index, including those that indirectly use it (that is, any referential constraints that rely on a primary key or unique constraint that uses the index are also dropped). Alternately, if you use the DROP CONSTRAINT option in an ALTER TABLE statement, NonStop SQL/MX will drop indexes that it created to implement that constraint.

Examples of DROP INDEX

- This example drops an index:

  ```
  DROP INDEX myindex;
  ```

DROP PROCEDURE Statement

Considerations for DROP PROCEDURE
Example of DROP PROCEDURE

The DROP PROCEDURE removes a stored procedure in Java (SPJ) from NonStop SQL/MX. To develop, deploy, and manage SPJs in SQL/MX, see the *SQL/MX Guide to Stored Procedures in Java*.

```
DROP PROCEDURE procedure-ref [RESTRICT|CASCADE]

procedure-ref is:
   [[catalog-name.]schema-name.]procedure-name
```

procedure-ref

specifies an ANSI logical name of the form:

[[*catalog-name*.]*schema-name*.]*procedure-name*

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

If you do not fully qualify the procedure name, NonStop SQL/MX qualifies it according to the current settings of CATALOG and SCHEMA. If you set the NAMETYPE attribute to NSK instead of ANSI and you do not fully qualify the procedure name, NonStop SQL/MX returns an error. For more information on the CATALOG, SCHEMA, and NAMETYPE attributes, see the System Defaults Table on page 10-37.

You cannot specify SQL parameters along with the procedure name. Each procedure name represents a unique SPJ in the database because NonStop SQL/MX does not support the overloading of procedure names.

RESTRICT

If you specify RESTRICT and the procedure is used in a trigger, the procedure is not dropped. The default is RESTRICT.

CASCADE

If you specify CASCADE and the procedure is used in a trigger, both the trigger and the procedure are dropped.

Considerations for DROP PROCEDURE

Authorization and Availability Requirements

To issue a DROP PROCEDURE statement, you must own the SPJ or be the super ID or schema owner.

Example of DROP PROCEDURE

- Drop an SPJ named ADJUSTSALARY from NonStop SQL/MX:

  ```
  DROP PROCEDURE samdbcat.persnl.adjustsalary;
  ```

DROP SCHEMA Statement

Considerations for DROP SCHEMA
Examples of DROP SCHEMA

The DROP SCHEMA statement deletes an SQL/MX schema and optionally deletes all objects within that schema. See Pseudocolumns on page 6-105.

```
DROP SCHEMA schema [CASCADE | RESTRICT]
```

Syntax Description of DROP SCHEMA

schema

is the name of the schema to drop.

If you specify RESTRICT, an error is reported if the specified schema is not empty.

If you specify CASCADE, all objects in the specified schema in addition to the schema itself, are dropped. The default is RESTRICT.

Considerations for DROP SCHEMA

Reserved Schemas

Schema names that start with DEFINITION_SCHEMA_VERSION_ are reserved for system metadata in all catalogs.

Schemas that start with NONSTOP_SQLMX_ are reserved for system metadata. These schema names are not reserved when used in a user-created catalog. Schemas named MXCS_SCHEMA in all catalogs are reserved for use by MXCS.

You cannot drop any of these reserved schemas or the objects contained in them.

Authorization and Availability Requirements

To drop a schema, you must own the schema or be the super ID. You must have remote passwords for any nodes to which the schema’s catalog has been registered. All the objects under the schema need not be owned by the schema owner to perform the DROP SCHEMA CASCADE operation.

Transaction Limits on DROP SCHEMA

If the schema is fairly large and contains many objects, DROP SCHEMA with the CASCADE option might fail with file system error 35, “Unable to obtain an I/O process control block, or the transaction or open lock unit limit has been reached.” In this case, too many locks were requested. When this occurs, you need to update MaxLocksPerTCB to 10000 or more.

In the Guardian environment, update through the SCF facility:

```
SCF

1-> info $<volume>,detail
STORAGE - Detailed Information Magnetic DISK
\<node>.$<volume> Common Disk Configuration Information:
*BackupCpu............................ 3
*HighPin.............................. ON
*PrimaryCpu........................... 2
*Program..............................$SYSTEM.SYSTEM.TSYSDP2
*StartState........................... STARTED

Disk Type Specific Information:
*AuditTrailBuffer/SQLMXBuffer (MB)..... 0
*AutoRevive........................... OFF
*AutoSelect........................... n/a
*AutoStart............................ ON
*CBPoolLen............................ 1000

*FSTCaching........................... OFF
*FullCheckpoints...................... ENABLED
*HaltOnError.......................... 1
*LKIDLongPoolLen...................... 8
*LKTableSpaceLen...................... 15
*MaxLocksPerOCB....................... 5000
*MaxLocksPerTCB....................... 5000
*NonAuditedInsert..................... OFF
More text? ([Y],N) n

2-> alter $volume,maxlockspertcb 10000
3-> info $volume,detail
STORAGE - Detailed Information Magnetic DISK \node.$volume
 Common Disk Configuration Information:
*BackupCpu............................ 3
*HighPin.............................. ON
*PrimaryCpu........................... 2
*Program..............................$SYSTEM.SYSTEM.TSYSDP2
*StartState........................... STARTED
```

```
Disk Type Specific Information:
*AuditTrailBuffer/SQLMXBuffer (MB)..... 0
*AutoRevive............................ OFF
*AutoSelect............................ n/a
*AutoStart............................. ON
*CBPoolLen............................. 1000
*FSTCaching............................ OFF
*FullCheckpoints....................... ENABLED
*HaltOnError........................... 1
*LKIDLongPoolLen....................... 8
*LKTableSpaceLen....................... 15
*MaxLocksPerOCB........................ 5000
*MaxLocksPerTCB........................ 10000
*NonAuditedInsert...................... OFF
More text? ([Y],N) n
```

Examples of DROP SCHEMA

- This example drops an empty schema:

  ```
  DROP SCHEMA sales RESTRICT;
  ```

DROP SEQUENCE Statement

Considerations for DROP SEQUENCE

Examples of DROP SEQUENCE

The DROP SEQUENCE statement drops a sequence generator from the current schema. The operation removes all the USAGE privileges granted for the sequence generator.

```
DROP SEQUENCE sequence
```

Syntax Description of DROP SEQUENCE

sequence

specifies the ANSI name of the sequence generator.

Considerations for DROP SEQUENCE

Authorization Requirements

The schema owner, Super ID or the object owner can execute a DROP SEQUENCE statement.

Restrictions

The ANSI CASCADE option is not supported for sequence generators.

Recovery

Starting with SQL/MX Release 3.2.1, the SAVE_DROPPED_TABLE_DDL CQD supports sequence generators. If you set this CQD to ON before executing a DROP SEQUENCE statement, the DDL of the sequence generator is stored in an OSS file. The DDL text is saved only for sequence generators that are explicitly dropped using a DROP SEQUENCE statement and not for sequence generators implicitly dropped using a DROP SCHEMA ... CASCADE operation.

Examples of DROP SEQUENCE

This example drops a sequence generator:

```
DROP SEQUENCE myseq;
```

DROP SQL Statement

The DROP SQL statement puts NonStop SQL/MX in an uninitialized state. It drops a system catalog that has no user-created catalogs.

DROP SQL is an SQL/MX extension.

```
DROP SQL
```

Considerations for DROP SQL

After you run the DROP SQL statement, you must run the InstallSqlmx script again to re-enable the use of SQL on the system. See the *SQL/MX Installation and Management Guide* for a description of this script.

Before executing DROP SQL, you must uninitialize NonStop MXCS. See the *SQL/MX Connectivity Service Manual* for details on this procedure.

Authorization and Availability Requirements

Only a SUPER user can execute this command. See Ownership on page 6-12. You must drop all user catalogs before performing the DROP SQL statement.

Examples of DROP SQL

- This example drops SQL on the system:

  ```
  DROP SQL;
  ```

DROP SQLMP ALIAS Statement

Considerations for DROP SQLMP ALIAS
Examples of DROP SQLMP ALIAS

The DROP SQLMP ALIAS statement is used to drop mappings from ANSI names to physical names of SQL/MP tables or views.

DROP SQLMP ALIAS is an SQL/MX extension.

```
DROP SQLMP ALIAS catalog.schema.object
```

catalog.schema.object

is the alias name of an SQL/MP table or view. *catalog* and *schema* denote ANSI-defined catalog and schema, and *object* is a simple name for the table or view. If any of the three parts of the name is an SQL/MX reserved word, you must delimit it by enclosing it in double quotes. For example: mycat."sql".myview.

See Catalogs on page 6-3 and Pseudocolumns on page 6-105.

Considerations for DROP SQLMP ALIAS

Usage Restrictions

If no alias exists for a given logical name, NonStop SQL/MX returns an error.

Any applications that attempt to use the dropped mapping will get an error because the specific alias no longer exists.

The DROP SQLMP ALIAS statement does not cause the underlying SQL/MP object to be dropped. Similarly, dropping an underlying SQL/MP object does not cause any SQLMP aliases to be dropped. Those aliases remain unchanged and orphaned.

Authorization and Availability Requirements

To drop an alias, you must be the owner of the schema in which the alias resides or be the super ID.

Examples of DROP SQLMP ALIAS

- Suppose that you have created an SQL/MP table by using this SQL/MP CREATE TABLE statement:

```
CREATE TABLE $myvol.mysubvol.mytable
    ( num    NUMERIC (4) UNSIGNED NOT NULL
     ,name    VARCHAR (20)
     ,PRIMARY KEY (num) );
```

This statement creates a mapping in the system metadata table:

```
CREATE SQLMP ALIAS mycatalog.myschema.mytable
                   $myvol.mysubvol.mytable;
```

This statement drops the mapping in the system metadata table:

```
DROP SQLMP ALIAS mycatalog.myschema.mytable;
```

DROP TABLE Statement

Considerations for DROP TABLE
Examples of DROP TABLE

The DROP TABLE statement deletes an SQL/MX table and any indexes, constraints, and inactive locks on the table. See Database Object Names on page 6-13.

```
DROP TABLE table [RESTRICT| CASCADE]
```

Syntax Description of DROP TABLE

table

is the name of the table to delete. If the table has an active DDL lock, neither the table nor any of its dependent objects are dropped. If you specify RESTRICT and *table* is referenced by a view, a trigger, or a referential constraint of another table, or if the table has an active DDL lock, the specified table cannot be dropped. If you specify CASCADE, the table and all of its views, triggers, referential constraints, and inactive DDL locks are dropped.

A table that has an active DDL lock (one for which the process that created it still exists) cannot be dropped even if you specify CASCADE. An active DDL lock is released when the utility locking the file completes.

The default is RESTRICT.

Considerations for DROP TABLE

Restrictions

You can drop a table with partitions, but you cannot drop individual partitions within a table with the DROP TABLE statement. However, you can drop these partitions by using the MODIFY utility. See MODIFY Utility on page 2-271.

You cannot drop an SQL/MP table by using its SQL/MP alias name.

Authorization and Availability Requirements

To drop a table, you must own the schema which contains the table or be the super ID or object owner. The associated objects can be dropped using the CASCADE option.

Recovery

When a table is dropped, NonStop SQL/MX automatically saves the DDL needed to re-create the table in an OSS file. If you do not want to save this text, set the SAVE_DROPPED_TABLE_DDL control query to "OFF".

The DDL is saved so that you can later retrieve it if you need to re-create the dropped table for any reason. If the table needs to be recovered with TMF or re-created for use in an RDF backup database, full Guardian file names are preserved and can be used to create identical file names.

Grant and revoke privileges are saved as part of the DDL text.

DDL text is saved only for user base tables that are explicitly dropped with a DROP TABLE statement. DDL text is not saved for tables that are implicitly dropped as a result of DROP SCHEMA CASCADE or DROP TRIGGER statements (that is, dropping the trigger temporary tables).

For details on this control query default, see Table Management on page 10-81.

Examples of DROP TABLE

- This example drops a table:

```
DROP TABLE mycat.mysch.mytable RESTRICT;
```

DROP TRIGGER Statement

Considerations for DROP TRIGGER
Examples of DROP TRIGGER

The DROP TRIGGER statement is used to drop a trigger on an SQL/MX table.

```
DROP TRIGGER trigger-name;
```

Syntax Description of DROP TRIGGER

trigger-name

specifies the ANSI logical name of the trigger to be dropped, of the form:

```
[[catalog-name.]schema-name.]trigger-name
```

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

Considerations for DROP TRIGGER

Authorization and Availability Requirements

To drop a trigger, you must own its schema or be the super ID or object owner.

Examples of DROP TRIGGER

- This example drops a trigger:

```
DROP TRIGGER my-trigger;
```

DROP VIEW Statement

Considerations for DROP VIEW
Examples of DROP VIEW

The DROP VIEW statement deletes an SQL/MX view. See Views on page 6-115.

```
DROP VIEW view [CASCADE | RESTRICT]
```

Syntax Description of DROP VIEW

view

is the name of the view to drop. If you specify RESTRICT, you cannot drop the specified view if it is referenced in the query expression of any other view or in the search condition of another object's constraint. If you specify CASCADE, any such dependent objects are dropped. The default is RESTRICT.

Considerations for DROP VIEW

Authorization and Availability Requirements

To drop a view, you must own the schema that contains the view or be the super ID or object owner.

Examples of DROP VIEW

- This example drops a view:

```
DROP VIEW mycat.mysch.myview RESTRICT;
```

DUP Utility

[Considerations for DUP](#)
[Examples of DUP](#)

DUP is a syntax-based utility that can be executed through MXCI. The DUP utility creates a copy of an SQL/MX table and optionally its indexes and constraints.

```
DUP source-target-list [ mapping ][,dup-option
    [, dup-option]...]

source-target-list is:
   { source-table TO target-table }

source-table is:
  [[catalog].schema.]object

target-table is:
  [{catalog | *}.{schema | *}.]object

mapping is:
  { LOCATION (volume-map [, volume-map ] ...)  }

volume-map is:
  [PART] [\node.]volume TO [\node.] $volume

dup-option is:
   { TARGET { NEW | PURGE }
   | INDEX[ES] [ { [ON] | OFF | [ON] (index-list )}]
   | CONSTRAINT[S] [{[ON] (constraint-list) | OFF | ON}]
   | OUTFILE oss-file [CLEAR] }

index-list is:
  index-map [, index-map] ...

index-map is:
  source-index TO target-index [mapping]

constraint-list is:
  constraint-map [, constraint-map] ...

constraint-map is:
  source-constraint TO target-constraint

source-index is:  object
target-index is:  object

source-constraint is:   object
target-constraint is:   object
```

Syntax Description of DUP

source-target-list

specifies the *source-table* and *target-table* names.

source-table

specifies the ANSI name for the table to be copied. The form of the name is *catalog-name.schema-name.table-name*, where each part is an SQL identifier. If you do not specify the catalog and schema parts of the *source-table*, DUP uses the default catalog and schema values for that session.

target-table

specifies the name of the target table. An asterisk (*) in the catalog or schema part of the target object name indicates copying the corresponding position of the source object name. If you do not specify a catalog and schema, DUP uses the corresponding catalog and schema of the source table, similar to the asterisk (*) option.

The name of the target object must be different from the name of the source object.

An error is returned if the source catalog, source schema, source object, target catalog, or target schema does not exist or if the target table is the same as the source table. DUP does not support duplication of views.

mapping

specifies which volumes DUP uses for the target partitions of tables and their dependent indexes. If you do not specify the *mapping* option, target partitions are mapped to the same volumes as the source partition's counterpart.

dup-option

specifies the different DUP options available for the operation, including:

`TARGET {NEW | PURGE}`

`TARGET` is an optional clause that specifies the action if the *target-table* already exists. `NEW` specifies that a new target table be created. If the target table already exists, an error is returned. `PURGE` specifies that the target table, if it exists, should be dropped and a new target table created. If the target table does not exist, a new target table is created. The default is `NEW`.

Caution. If you choose the PURGE option, DUP first drops the target table. If an error occurs further along in the DUP operation, you cannot recover the original target table. You should back up the target table before you begin your DUP operation.

`INDEX[ES] [{ON [(`*`index-list`*`)] | OFF }]`

is an optional clause that specifies how dependent indexes are copied. `INDEX[ES]` is the same as `INDEX[ES] ON`. Unpopulated indexes are not duplicated.

`ON [(`*`index-list`*`)]`

duplicates all user-created and system-generated indexes. If the source and target tables reside in the same catalog and schema, you must specify *`index-list`* to identify the new index names. The default is `ON`.

System generated indexes are copied only if the `CONSTRAINTS` option is `ON`.

`index-list`

is a list of *`index-maps.`* This option duplicates only a subset of the indexes available on the source table. If you specify this option, only those indexes that are described in the *`index-list`* are duplicated. If an index that exists on the source table does not have corresponding item in the *`index-list`*, DUP does not duplicate the index.

`index-map`

You cannot define names for system-generated indexes. If you do not specify *`index-map`*, names are generated for all indexes.

You can specify a *`mapping`* clause for each pair in *`index-list`*. If you do not specify a mapping clause as part of *`index-map`*, the *`mapping`* clause for the base table is used. If you do not specify a *`mapping`* clause, the source partition volume is used for target partitions. If you specify a nonexistent index name in the index map, DUP returns an error.

Mapping can be at the table level or the index level. If you specify mapping at the table level, DUP uses that mapping. If the index does not have a corresponding volume mapping, DUP uses the same volume as the source index to create the target index partitions.

`OFF`

does not duplicate indexes. Constraints that requires an index are not duplicated, including unique constraints and non clustering primary key constraints. Referential integrity constraints are never duplicated. If `CONSTRAINTS` is also `OFF`, the DUP operation proceeds.

`CONSTRAINT[S] [{[ON] (`*`constraint-list`*`) | OFF | ON}]`

is an optional clause that specifies whether to copy DROPPABLE constraints not null, primary key, and check. If you do not specify this clause, the default is CONSTRAINTS ON. UNIQUE constraints are treated as DROPPABLE constraints and are duplicated if you specify CONSTRAINTS ON. NOT DROPPABLE constraints are always duplicated.

`ON`

duplicates all DROPPABLE unique, not null, primary key, and check constraints. The default is `ON`.

If the source table has unique or droppable primary key constraints, the INDEX option must be set to ON, otherwise DUP returns an error.

`constraint-list`

is a list of *`constraint-maps`*.

`constraint-map`

specifies the target constraint name. If you do not specify *`constraint-map`*, DUP uses a generated name based on the target table name for each constraint.

`source-constraint`

is the constraint to be duplicated, including unique, not null, primary key, and check constraints.

`target-constraint`

is the constraint that is formed after the DUP operation. If you specify *`constraint-map`*, *`target-constraint`* is required.

`OFF`

does not copy constraints. If `indexes` is ON, only user-created indexes are copied.

`log-clause`

specifies logging functionality to the DUP function and starts logging to a disk file. While logging is in progress, the DUP commands that are entered are executed and written to the disk file. The output of the DUP command is also written to the disk file.

`OUTFILE` *`oss-file`* `[CLEAR]`

specifies that the output go to a disk file. *`oss-file`* is the path name of the file to which DUP writes commands and command output. CLEAR clears the *`oss-file`* before logging begins. If CLEAR is omitted, OUTFILE appends the new log to existing data in *`oss-file`*.

`oss-file` cannot contain the “,” (comma) character or the “;” (semicolon) character.

Considerations for DUP

- Referential integrity constraints and triggers are ignored.
- The source table can exist on a remote node and be referenced by the current DUP operation if the remote node is visible to the local node. The target table can also exist in a catalog and schema that reside on a visible remote node.
- DUP does not check disk space before running the request. You must confirm that enough disk space is available before running the DUP request.
- DUP displays errors if the source table or target table and its dependent indexes cannot be accessed, or if the load fails in response to a resource or file system problem.

  You must run the RECOVER utility to clean up a failed DUP operation. If the DUP operation fails after all of the data is successfully copied to the target objects, specify RECOVER with the RESUME option to complete the DUP operation. If the DUP operation fails before the data is successfully copied, specify RECOVER with the CANCEL option to roll back the DUP operation. This status can be found by reading the DDL_LOCKS definition schema table in the source table's catalog. If you run the RECOVER operation with the incorrect option, RECOVER displays an error message so you can rerun it with the correct option. For details, see Checking DDL Locks on page 2-9.

  No restart facility is available to handle partially copied data.
- During the DUP operation, the target table is marked as corrupt to prevent other processes from viewing the data until the operation completes successfully.
  - All utility operations have the potential to run for hours, especially those that involve a great deal of data movement. To manage systems effectively, you need to know how far the operation has proceeded and how much longer it needs to run. Utilities provide progress reports that indicate what step is in progress. Utility operations periodically place progress reports in the metadata tables through the DDL lock mechanism. You can examine the metadata to get the latest information. These reports are referred to as the operation's progress. The DUP operation has the option to log these progress reports to an OSS text file. DDL locks

    Many utility operations run in multiple TMF transactions. As a result, conflicting operations that change metadata and label information affecting the outcome of the utility are executed concurrently.

    To serialize these utility operations, NonStop SQL/MX has the concept of a DDL lock. This is a lock that prevents database structure changes from occurring while a utility request is executing. A utility request informs SQL that it is running, perform commands in as many transactions as necessary, then informs SQL that the operation has completed. While the utility request is running, no conflicting DDL or utility operation can occur. That is, you can make no database structural change that would affect the utility.

DDL lock information is persistent across transaction boundaries. If the utility operation fails unexpectedly such as a process failure, you must run the RECOVER utility to remove the lock and clean up the operation. When you run the RECOVER utility, DDL lock information is retrieved and the correct clean up operation is performed. If you run the RECOVER operation with the incorrect option, RECOVER displays an error message so you can rerun it with the correct option.

While the operation is proceeding, you can select state information from metadata tables to determine the utility progress. If the operation terminates unexpectedly, you can also select this information to determine where the operation failed.

- DUP records operation progress steps in the DDL_LOCKS metadata table. You can query this table to determine the DUP operation's progress:

DUP Operation Step	**Step Progress Status**
Step 1	DDL lock has been created.
Step 2	Target table has been created.
Step 3	Source table is open.
Step 4	All source objects are open.
Step 5	Target table is open.
Step 6	All target objects are open.
Step 7	All table partitions are copied.
Step 8	All index partitions are copied.
Step 9	All objects for *`catalog.schema`*.table have been copied.*
Step 10	Target object is now available (corrupt attribute is turned off, audit attribute is turned on).
Step 11	DDL lock is removed.

* Any process, CPU, or system failure that occurs before this point causes the operation to be rolled back. Any failure after this point can be resumed.

- An error is returned if a user transaction exists.
- An error is returned if a DUP operation is attempted on an SQL/MX metadata table (histogram, system defaults, or MXCS tables).
- DUP does not support RI constraints duplication.
- For a source table that contains an IDENTITY column, the DUP utility duplicates the source table IDENTITY column attributes and the corresponding SG Attributes onto the target table. DUP copies the data of the source table onto the target table. DUP also copies the source SG Table `CURRENT_VALUE` onto the target SG Table `CURRENT_VALUE`.

Examples of DUP

- This example copies the partitions of the source table (using a different catalog and schema) to the same locations:

```
DUP mycat.myschema.mytable1 TO mycat1.myschema1.*;
```

- This example copies the partitions of the source table on `$data1` and `$data2` to the partitions of the target table on `$data2` and `$data3` respectively. If there is no PART clause for a specific volume and source partitions exist on that volume, the target partitions are created on the same volume as the source partitions.

```
DUP mycat.myschema1.mytable TO *.myschema2.*
LOCATION (PART $data1 TO $data2, PART $data2 TO $data3);
```

- This example copies the partitions of the source table to the same locations. The target table, if it exists, is dropped, and a new one is created.

```
DUP mycat1.myschema.mytable TO mycat2.*.*,TARGET PURGE;
```

EXECUTE Statement

Considerations for EXECUTE
MXCI Examples of EXECUTE
C Examples of EXECUTE
COBOL Examples of EXECUTE

The EXECUTE statement executes an SQL statement previously compiled by a PREPARE statement. You can use EXECUTE in an MXCI session or in an embedded SQL program.

C/COBOL

Input data can be supplied either by using host variables or through an SQL descriptor area in an embedded SQL program. Similarly, you can place output data either directly into host variables or into an SQL descriptor area. For more information, see the *SQL/MX Programming Manual for C and COBOL*. ■

MXCI

```
EXECUTE statement-name
   [USING param-value [,param-value]...] ■
```

C/COBOL

```
EXECUTE statement-name
   [USING {argument-list | descriptor-spec}]
   [INTO {argument-list | descriptor-spec}]

statement-name is:
   statement-name | ext-statement-name

argument-list is:
   variable-spec [,variable-spec]...

descriptor-spec is:
   SQL DESCRIPTOR descriptor-name

ext-statement-name is:
   [GLOBAL | LOCAL] value-specification

variable-spec is:
   :variable-name [[INDICATOR] :indicator-name]

descriptor-name is:
   [GLOBAL | LOCAL] value-specification ■
```

`statement-name`

is the name of a prepared SQL statement—that is, the statement name used in the PREPARE statement. `statement-name` is an SQL identifier. See Identifiers on page 6-56.

C/COBOL

The statement name is not case-sensitive in MXCI. ■

The statement name is case-sensitive in embedded SQL—for example, the statement named `findemp` is not equivalent to the statement named `FINDEMP`.

The module that contains EXECUTE must also contain a PREPARE statement for *statement-name*. ■

MXCI `USING param-value [,param-value]...`

specifies values for the unnamed parameters in the prepared statement. The data type of a parameter value must be compatible with the data type of the associated parameter. Parameter values are substituted for unnamed parameters in the prepared statement by position—the i-th value in the USING clause is the value for the i-th unnamed parameter in the statement.

If there are more values in the value list than there are unnamed parameters in the PREPARE statement, NonStop SQL/MX ignores the extra values; if there are fewer unnamed values in the value list, NonStop SQL/MX returns an error.

The values for any named parameters in the prepared statement must be previously specified with SET PARAM commands. ■

MXCI `param-value`

is a numeric or character literal that specifies the value for the parameter. The *param-value* can also be the NULL keyword. You must enter it in uppercase letters. If *param-value* is a character literal and the target column is character, you do not have to enclose it in single quotation marks. Its data type is determined from the data type of the column to which the literal is assigned. ■

C/COBOL *ext-statement-name*

is a *value-specification*—a host variable with a character data type. When EXECUTE executes, the content of the value specification must identify a statement previously prepared within the scope of EXECUTE. ■

C/COBOL `GLOBAL | LOCAL`

specifies the scope of the prepared statement. The default is LOCAL. A GLOBAL prepared statement can be executed within the SQL session. A LOCAL prepared statement can be executed only within the module or compilation unit in which it was prepared.

A prepared SQL statement must be currently available whose name is the value of *ext-statement-name* and whose scope is the same scope as specified in the EXECUTE statement. ■

C/COBOL `{USING | INTO}` *variable-spec* `[,`*variable-spec*`]...`

identifies the host variables for the parameters of *SQL-statement-name*.

Before EXECUTE with USING executes, the application must store information for each input parameter of the prepared statement in the appropriate host variable.

When EXECUTE with INTO executes, NonStop SQL/MX stores information into the host variables (and optionally their indicator variables) that correspond to columns specified in the select list for the prepared statement.

`:`*`variable-name`* `[[INDICATOR] :`*`indicator-name`*`]`

is a variable specification—a host variable with optionally an indicator variable. A variable name begins with a colon (:).

The data type of an indicator variable is exact numeric with a scale of 0. If the data returned in the host variable is null, the indicator parameter is set to a value less than zero. If character data returned is truncated, the indicator parameter is set to the length of the string in the database. ■

C/COBOL `{USING | INTO} SQL DESCRIPTOR` *`descriptor-name`*

identifies the SQL descriptor area for the parameters of *`SQL-statement-name`*. An SQL descriptor area must be currently allocated whose name is the value of *`descriptor-name`* and whose scope is the same scope specified in the EXECUTE statement.

Before EXECUTE with USING executes, the application must store information for each input parameter of the prepared statement in the descriptor area. Each parameter has an item descriptor.

When EXECUTE with INTO executes, NonStop SQL/MX stores information into the descriptor area about each column specified in the select list for the prepared statement. Each column has an item descriptor.

`descriptor-name`

is a *`value-specification`*—a character literal or host variable with character data type. When EXECUTE executes, the content of the value specification (if a host variable) gives the name of the descriptor area. ■

See An operation is a postfix merge if the range of data ends at the bottom of the partition. You can specify only the TO NEXT PARTITION clause. The split partition cannot be the last partition (the rightmost partition in the list). on page 2-279, SET PARAM Command on page 4-63, and MXCI Parameters on page 6-77.

Considerations for EXECUTE

Scope of EXECUTE

A statement must be compiled by PREPARE before you EXECUTE it, but after it is compiled, you can execute the statement multiple times without recompiling it.

MXCI The statement must have been compiled during the same MXCI session as its execution. ■

C/COBOL The statement must have been prepared during the same compilation unit as its execution. ■

MXCI Examples of EXECUTE

- Use PREPARE to compile a statement once, and then execute the statement multiple times with different parameter values. This example uses the SET PARAM command to set the parameter values in the prepared statement.

```
PREPARE FINDEMP FROM
     SELECT * FROM persnl.employee
     WHERE salary > ?SALARY AND jobcode = ?JOBCODE;
--- SQL command prepared.

SET PARAM ?SALARY 40000.00;
SET PARAM ?JOBCODE 450;
EXECUTE FINDEMP;

EMPNUM  FIRST_NAME  LAST_NAME   DEPTNUM  JOBCODE  SALARY
------  ----------  ----------  -------  -------  --------
   232  THOMAS      SPINNER        4000      450  45000.00
--- 1 row(s) selected.

SET PARAM ?SALARY 20000.00;
SET PARAM ?JOBCODE 300;
EXECUTE FINDEMP;

EMPNUM  FIRST_NAME  LAST_NAME   DEPTNUM  JOBCODE  SALARY
------  ----------  ----------  -------  -------  --------
    75  TIM         WALKER         3000      300  32000.00
    89  PETER       SMITH          3300      300  37000.40
   ...
--- 13 row(s) selected.
```

- Use EXECUTE USING for both parameter values, which are unnamed in the prepared statement:

```
PREPARE FINDEMP FROM
     SELECT * FROM persnl.employee
     WHERE salary > ? AND jobcode = ?;

EXECUTE FINDEMP USING 40000.00,450;
EXECUTE FINDEMP USING 20000.00,300;
```

- Use EXECUTE USING for one parameter value, which is unnamed in the prepared statement:

```
PREPARE FINDEMP FROM
     SELECT * FROM persnl.employee
     WHERE salary > ?SALARY AND jobcode = ?;

SET PARAM ?SALARY 40000.00;
EXECUTE FINDEMP USING 450;

SET PARAM ?SALARY 20000.00;
EXECUTE FINDEMP USING 300;
```

C Examples of EXECUTE

- Prepare and execute an UPDATE statement with dynamic input parameters:

```
...
strcpy(stmt_buffer,"UPDATE SALES.CUSTOMER"
          " SET CREDIT = ?"
          " WHERE CUSTNUM = CAST(? AS NUMERIC(4) UNSIGNED)")
...
EXEC SQL PREPARE upd_cust FROM :stmt_buffer;
...
/* Input values for parameters into host variables */
scanf("%s",in_credit);
scanf("%ld",&in_custnum);
...
EXEC SQL EXECUTE upd_cust USING :in_credit, :in_custnum;
...
```

- Prepare a statement, allocate input and output descriptor areas, describe the input and output descriptor areas, and execute the statement by using the descriptor areas:

```
...
strcpy(stmt_buffer,"SELECT * FROM EMPLOYEE"
       " WHERE EMPNUM = CAST(? AS NUMERIC(4) unsigned)");
...
EXEC SQL PREPARE S1 FROM :stmt_buffer;
...
desc_max = 1;
EXEC SQL ALLOCATE DESCRIPTOR 'in_args' WITH MAX :desc_max;
desc_max = 6;
EXEC SQL ALLOCATE DESCRIPTOR 'out_cols' WITH MAX :desc_max;
...
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR 'in_args';
EXEC SQL DESCRIBE OUTPUT S1 USING SQL DESCRIPTOR 'out_cols';
...
EXEC SQL EXECUTE S1 USING SQL DESCRIPTOR 'in_args'
                 INTO SQL DESCRIPTOR 'out_cols';
...
```

- This example uses extended statement names:

```
...
strcpy(stmt,"ins_cust1");
EXEC SQL PREPARE :stmt FROM :stmt_buffer;
EXEC SQL EXECUTE :stmt;
...
strcpy(stmt,"ins_cust2");
EXEC SQL PREPARE :stmt FROM :stmt_buffer;
EXEC SQL EXECUTE :stmt;
```

COBOL Examples of EXECUTE

- Prepare and execute an UPDATE statement with dynamic input parameters:

```
     ...
     MOVE "UPDATE SALES.CUSTOMER SET CREDIT = ?
        & " WHERE CUSTNUM = CAST(? AS NUMERIC(4) UNSIGNED)"
     TO stmt-buffer.
     ...
     EXEC SQL PREPARE upd_cust FROM :stmt-buffer END-EXEC.
     ...
* Input values for parameters into host variables
     ACCEPT in-credit.
     ...
     ACCEPT in-custnum.
     ...
     EXEC SQL EXECUTE upd_cust
              USING :in-credit, :in-custnum
     END-EXEC.
     ...
```

- Prepare a statement, allocate input and output descriptor areas, describe the input and output descriptor areas, and execute the statement by using the content of the descriptor areas:

```
...
MOVE "SELECT * FROM EMPLOYEE"
   & " WHERE EMPNUM = CAST(? AS NUMERIC(4) UNSIGNED)"
TO stmt-buffer.
...
EXEC SQL PREPARE S1 FROM :stmt-buffer END-EXEC.
...
MOVE 1 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR 'in_args'
           WITH MAX :desc-max END-EXEC.
MOVE 6 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR 'out_cols'
           WITH MAX :desc-max END-EXEC.
...
EXEC SQL DESCRIBE INPUT S1
         USING SQL DESCRIPTOR 'in_args'
END-EXEC.
EXEC SQL DESCRIBE OUTPUT S1
         USING SQL DESCRIPTOR 'out_cols'
```

```
END-EXEC.
...
EXEC SQL EXECUTE USING SQL DESCRIPTOR 'in_args'
                 INTO SQL DESCRIPTOR 'out_cols'
END-EXEC.
...
```

- This example uses extended statement names:

```
...
MOVE "ins_cust1" TO stmt.
EXEC SQL PREPARE :stmt FROM :stmt-buffer END-EXEC.
EXEC SQL EXECUTE :stmt END-EXEC.
...
MOVE "ins_cust2" TO stmt.
EXEC SQL PREPARE :stmt FROM :stmt-buffer END-EXEC.
EXEC SQL EXECUTE :stmt END-EXEC.
```

EXPLAIN Statement

Considerations for EXPLAIN

Examples of EXPLAIN

The EXPLAIN statement generates and displays the result of the EXPLAIN function, describing an access plan for a SELECT, INSERT, DELETE, UPDATE, or CALL statement. It displays the query execution plans in a readable format. It can display plans from modules created by the SQL/MX compiler.

The EXPLAIN statement can also be used from JDBC or ODBC application like any other SQL/MX Statement.

For a description of the result table of the EXPLAIN function, see EXPLAIN Function on page 8-66.

You can use the EXPLAIN statement within an MXCI, JDBC, or ODBC session.

```
EXPLAIN [options {'f' | 'n' | 'e' | 'm'}] {query-text |
prepared-stmt-name | 'stmt-name' from 'module-name'}
```

`f`

formatted.

`n`

normal user (default setting).

`e`

expert user.

`m`

machine readable format.

query-text

is a DML statement such as SELECT * FROM T3.

`prepared-stmt-name`

is an SQL identifier containing the name of a statement already prepared in this session. An SQL identifier is not case sensitive unless it is double-quoted. It must be double-quoted if it contains blanks, lower case letters, or special characters; normally they are not required. It must start with a letter.

module-name

is the name of a file where a static compile stores the information. It is specified within single quotes.

stmt-name

is the statement pattern that includes name or %, or S%, and so on with single quotes.

The syntax for the EXPLAIN statement supports four output options. Table 2-4 summarizes the options.

Table 2-4. EXPLAIN Statement Options

Syntax	Option Type	Purpose
`OPTIONS 'f'`	Formatted	Provides the basic information contained in the query execution plan. This information is formatted for readability and limited to 79 characters (one line) per operator.
`OPTIONS 'n'`	Normal user	Provides the most important information contained in the query execution plan. This information is formatted for readability and is the default output format.
`OPTIONS 'e'`	Expert user	Provides all the information contained in the query execution plan. This information is formatted for user readability.
`OPTIONS 'm'`	Machine readable	Provides all the information contained in the query execution plan. This information is formatted for machine readability (easy to parse with software tools).

For more information about the operators in the query execution plan, see the *SQL/MX Query Guide*.

Considerations for EXPLAIN

Case Considerations

In most cases, words in the commands can be in uppercase or lowercase. The letter following the OPTIONS keyword must be within single quotes and in lowercase.

Number Considerations

Costs are given in a generic unit of effort. They show relative costs of an operation.

When numbers are displayed as 0.01 for `OPTIONS 'n'` (or 0.0001 for `OPTIONS 'e'`), the numbers have likely been rounded up. However, if the numbers are zero, the display shows “0”.

When trailing decimal digits are zero, they are dropped. For example, 6.4200 will be displayed as 6.42 and 5.0 will be displayed as 5, without a decimal point.

Machine-readable [`OPTIONS 'm'`] Considerations

The machine-readable option provides an output in the format that can be read only by machines but is suitable for programs.Table 2-5 lists the fields of the OPTIONS 'm' output.

Table 2-5. Fields of OPTIONS 'm' Output

Column name	Data Type	Description
MODULE NAME	CHAR (60)	Reserved for future use.
STATEMENT NAME	CHAR (60)	Statement name; truncated on the right if longer than 60 characters.
PLAN_ID	INT	Unique system-generated plan ID automatically assigned by SQL; generated at compile time.
SEQ_NUM	CHAR (30)	Sequence number of the current operator in the operator tree; indicates the sequence in which the operator tree is generated.
OPERATOR	CHAR (30)	Current operator type.
LEFT_CHILD_SEQ_NUM	INT	Sequence number for the first child operator of the current operator; displays NULL if the operator has no child operators.
RIGHT_CHILD_SEQ_NUM	INT	Sequence number for the second child operator of the current operator; displays NULL if the operator does not have a second child.
TNAME	CHAR (60)	For operators in a scan group, the full name of base table is truncated on the right if it is too long for the column. If the correlation name differs from the table name put the correlation name first and then table name in parentheses.
CARDINALITY	REAL	Estimated number of rows that are returned by the current operator.
OPERATOR_COST	REAL	Estimated cost associated with the current operator to execute the operator.
TOTAL_COST	REAL	Estimated cost associated with the current operator to execute the operator, including the cost of all subtrees in the operator tree.
DETAIL_COST	VARCHAR (200)	Cost vector of five items, which are described in detail in Table 2-6, Cost Factors of DETAIL_COST column.
DESCRIPTION	VARCHAR (3000)	Additional information about the operator. For more information about the DESCRIPTION of all operators, see the *SQL/MX Query Guide*.

Table 2-6 lists the cost factors of the DETAIL_COST column:

Table 2-6. Cost Factors of DETAIL_COST column

Cost Factor	Description
CPU_TIME	An estimate of the number of seconds of processor time it might take to execute the instructions for this operator. A value of 1.0 is 1 second.
IO_TIME	An estimate of the number of seconds of I/O time (seeks plus data transfer) to perform the I/O for this operator.
MSG_TIME	An estimate of the number of seconds it takes for the messaging for this operator. The estimate includes the time for the number of local and remote messages and the amount of data sent.
IDLETIME	An estimate of the number of seconds to wait before an event. The estimate includes the amount of time to open a table or start an Executor Server Process (ESP) process.
PROBES	The number of times the operator will be executed. Usually, the value is 1, but it can be greater when you have, for example, an inner scan of a nested-loop join.

Examples of EXPLAIN

Consider a table ‘part’ with a unique index px1. Run the following commands:

```
Create table part  (
   p_partkey      INT                  not null not droppable,
   p_name         VARCHAR(55)          not null not droppable,
   p_mfgr         CHAR(25)             not null not droppable,
   p_brand        CHAR(10)             not null not droppable,
   p_type         VARCHAR(25)          not null not droppable,
   p_size         INT                  not null not droppable,
   p_container    CHAR(10)             not null not droppable,
   p_retailprice  NUMERIC(12,2)        not null not droppable,
   p_comment      VARCHAR(23)          not null not droppable,
PRIMARY KEY (p_partkey)  not droppable);


Create unique index px1 on part
(
     p_type
   , p_size
   , p_mfgr
   , p_brand
   , p_container
   , p_partkey
);
```

Run the following commands to create a table ‘partsupp’ and indexes psx1 and psx2:

```
Create table partsupp  (
   ps_partkey          INT               not null not droppable,
```

```
   ps_suppkey          INT              not null not droppable,
   ps_availqty         INT              not null not droppable,
   ps_supplycost       NUMERIC(12,2)    not null not droppable,
   ps_comment          VARCHAR(199)     not null not droppable,
PRIMARY KEY (ps_partkey,ps_suppkey)  not droppable);


Create index psx1 on partsupp
(
     ps_suppkey
   , ps_supplycost
   , ps_availqty
);


Create index psx2 on partsupp
(
     ps_partkey
   , ps_suppkey
   , ps_supplycost
   , ps_availqty
);
```

To use the EXPLAIN statement with a prepared statement, prepare the query, and then use the EXPLAIN statement:

```
prepare xx from
select * from part where p_partkey = (select max(ps_partkey)
from partsupp);
```

- Use OPTIONS 'f':

  ```
  >>explain options 'f' xx;
  ```

  The following output is displayed:

```
LC   RC   OP   OPERATOR              OPT       DESCRIPTION           CARD
---- ---- ---- --------------------  --------  ----------------  ---------
7    .    8    root                                              1.00E+000
4    6    7    nested_join                                       1.00E+000
5    .    6    partition_access                                  1.00E+000
.    .    5    file_scan_unique      fr        PART (s)          1.00E+000
3    .    4    partition_access                                  1.00E+000
2    .    3    shortcut_scalar_aggr                              1.00E+000
1    .    2    firstn                                            1.00E+000
.    .    1    index_scan                      PSX2 (s)          1.00E+002
--- SQL operation complete.
```

- Use OPTIONS 'e':

  ```
  >>explain options 'e' xx;
  ```

The following output is displayed:

```
------------------------------------------------------------------ PLAN
SUMMARY
MODULE_NAME .............. DYNAMICALLY COMPILED
STATEMENT_NAME ........... XX
PLAN_ID .................. 212122794829778687
ROWS_OUT ................. 1
EST_TOTAL_COST ........... 0.2332
STATEMENT ................ select *
                           from part
                           where p_partkey = (select max(ps_partkey) from
                           partsupp);


------------------------------------------------------------------ NODE
LISTING
ROOT ======================================  SEQ_NO 8      ONLY CHILD 7
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.196
EST_TOTAL_COST ........... 0.2332
  cpu_cost ............... 0.0016
  io_cost ................ 0.0372
  msg_cost ............... 0
  idle_cost .............. 0.196
  probes ................. 1
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  statement_index ........ 0
  xn_access_mode ......... read_only
  plan_version ....... 2,400
  SCHEMA ................. CAT.SCH
  select_list ............ CAT.SCH.PART.P_PARTKEY, CAT.SCH.PART.P_NAME,
                             CAT.SCH.PART.P_MFGR, CAT.SCH.PART.P_BRAND,
                             CAT.SCH.PART.P_TYPE, CAT.SCH.PART.P_SIZE,
                             CAT.SCH.PART.P_CONTAINER,
                             CAT.SCH.PART.P_RETAILPRICE,
                             CAT.SCH.PART.P_COMMENT
```

```
NESTED_JOIN ==============================  SEQ_NO 7      CHILDREN 4, 6
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.0001
EST_TOTAL_COST ........... 0.0372
  cpu_cost ............... 0.0016
  io_cost ................ 0.0372
  msg_cost ............... 0
  idle_cost .............. 0
  probes ................. 1
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner
  join_method ............ nested


PARTITION_ACCESS ==========================  SEQ_NO 6      ONLY CHILD 5
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.0008
EST_TOTAL_COST ........... 0.0206
  cpu_cost ............... 0.0008
  io_cost ................ 0.0206
  msg_cost ............... 0
  idle_cost .............. 0
  probes ................. 1
DESCRIPTION
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  buffer_size ....... 31,000
  record_length ........ 177
  space_usage ............ 13:8:8:36
  eid_space_computation    on


FILE_SCAN_UNIQUE ==========================  SEQ_NO 5      NO CHILDREN
TABLE_NAME ............... CAT.SCH.PART
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.0206
EST_TOTAL_COST ........... 0.0206
  cpu_cost ............... 0.0001
  io_cost ................ 0.0206
  msg_cost ............... 0
  idle_cost .............. 0
  probes ................. 1
DESCRIPTION
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. unique access of table CAT.SCH.PART
  object_type ............ CAT.SCH.PART
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock cursor
  access_mode ............ not specified, defaulted to read committed
  columns_retrieved ...... 9
  fast_replydata_move .... used
  key_columns ............ P_PARTKEY
  key .................... (P_PARTKEY = max(CAT.SCH.PSX2.PS_PARTKEY))
```

```
PARTITION_ACCESS ==========================  SEQ_NO 4      ONLY CHILD 3
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.0008
EST_TOTAL_COST ........... 0.0165
  cpu_cost ............... 0.0008
  io_cost ................ 0.0165
  msg_cost ............... 0
  idle_cost .............. 0
  probes ................. 1
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  buffer_size ....... 31,000
  record_length .......... 8
  space_usage ............ 12:8:8:32
  eid_space_computation    on


SHORTCUT_SCALAR_AGGR ======================  SEQ_NO 3      ONLY CHILD 2
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.0001
EST_TOTAL_COST ........... 0.0165
  cpu_cost ............... 0.0001
  io_cost ................ 0.0165
  msg_cost ............... 0
  idle_cost .............. 0
  probes ................. 1
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  aggregates ............. max(CAT.SCH.PSX2.PS_PARTKEY)


FIRSTN ====================================  SEQ_NO 2      ONLY CHILD 1
REQUESTS_IN .............. (not found)
ROWS_OUT ................. 1
EST_OPER_COST ............ 0
EST_TOTAL_COST ........... 0
  OPERATOR_COST .......... 0
  ROLLUP_COST ............ 0
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
```

```
INDEX_SCAN ================================  SEQ_NO 1      NO CHILDREN
TABLE_NAME ............... CAT.SCH.PARTSUPP
REQUESTS_IN .............. 1
ROWS_OUT ............... 100
EST_OPER_COST ............ 0.0206
EST_TOTAL_COST ........... 0.0206
  cpu_cost ............... 0.0001
  io_cost ................ 0.0206
  msg_cost ............... 0
  idle_cost .............. 0
  probes ................. 1
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. full scan of index
CAT.SCH.PSX2(CAT.SCH.PARTSUPP)
  scan_direction ......... reverse
  object_type ............ CAT.SCH.PARTSUPP
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock cursor
  access_mode ............ not specified, defaulted to read committed
  columns_retrieved ...... 6
  key_columns ............ CAT.SCH.PSX2.PS_PARTKEY,
CAT.SCH.PSX2.PS_SUPPKEY,
                             CAT.SCH.PSX2.PS_SUPPLYCOST,
                             CAT.SCH.PSX2.PS_AVAILQTY,
CAT.SCH.PSX2.PS_PARTKEY,
                             CAT.SCH.PSX2.PS_SUPPKEY
  begin_key .............. (CAT.SCH.PSX2.PS_PARTKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPLYCOST = <max>),
                             (CAT.SCH.PSX2.PS_AVAILQTY = <max>),
                             (CAT.SCH.PSX2.PS_PARTKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <max>)
  end_key ................ (CAT.SCH.PSX2.PS_PARTKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPLYCOST = <min>),
                             (CAT.SCH.PSX2.PS_AVAILQTY = <min>),
                             (CAT.SCH.PSX2.PS_PARTKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <min>)

--- SQL operation complete.
```

- Use `OPTIONS 'n'`:

```
>>explain options 'n' xx;
```

The following output is displayed:

```
------------------------------------------------------------------ PLAN
SUMMARY
MODULE_NAME .............. DYNAMICALLY COMPILED
STATEMENT_NAME ........... XX
PLAN_ID .................. 212122794829778687
ROWS_OUT ................. 1
EST_TOTAL_COST ........... 0.23
STATEMENT ................ select *
                           from part
                           where p_partkey = (select max(ps_partkey) from
                             partsupp);


------------------------------------------------------------------ NODE
LISTING
ROOT ======================================  SEQ_NO 8      ONLY CHILD 7
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.2
EST_TOTAL_COST ........... 0.23
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  statement_index ........ 0
  xn_access_mode ......... read_only
  plan_version ....... 2,400
  SCHEMA ................. CAT.SCH
  select_list ............ CAT.SCH.PART.P_PARTKEY, CAT.SCH.PART.P_NAME,
                             CAT.SCH.PART.P_MFGR, CAT.SCH.PART.P_BRAND,
                             CAT.SCH.PART.P_TYPE, CAT.SCH.PART.P_SIZE,
                             CAT.SCH.PART.P_CONTAINER,
                             CAT.SCH.PART.P_RETAILPRICE,
                             CAT.SCH.PART.P_COMMENT


NESTED_JOIN ===============================  SEQ_NO 7      CHILDREN 4, 6
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.01
EST_TOTAL_COST ........... 0.04
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner
  join_method ............ nested


PARTITION_ACCESS ==========================  SEQ_NO 6      ONLY CHILD 5
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.01
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  buffer_size ....... 31,000
  record_length ........ 177
  space_usage ............ 13:8:8:36
  eid_space_computation    on
```

```
FILE_SCAN_UNIQUE ==========================  SEQ_NO 5      NO CHILDREN
TABLE_NAME ............... CAT.SCH.PART
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.02
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. unique access of table CAT.SCH.PART
  object_type ............ CAT.SCH.PART
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock cursor
  access_mode ............ not specified, defaulted to read committed
  columns_retrieved ...... 9
  fast_replydata_move .... used
  key_columns ............ P_PARTKEY
  key .................... (P_PARTKEY = max(CAT.SCH.PSX2.PS_PARTKEY))


PARTITION_ACCESS ==========================  SEQ_NO 4      ONLY CHILD 3
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.01
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  buffer_size ....... 31,000
  record_length .......... 8
  space_usage ............ 12:8:8:32
  eid_space_computation    on


SHORTCUT_SCALAR_AGGR ======================  SEQ_NO 3      ONLY CHILD 2
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.01
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  aggregates ............. max(CAT.SCH.PSX2.PS_PARTKEY)


FIRSTN ====================================  SEQ_NO 2      ONLY CHILD 1
REQUESTS_IN .............. (not found)
ROWS_OUT ................. 1
EST_OPER_COST ............ 0
EST_TOTAL_COST ........... 0
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
```

```
INDEX_SCAN ================================  SEQ_NO 1      NO CHILDREN
TABLE_NAME ............... CAT.SCH.PARTSUPP
REQUESTS_IN .............. 1
ROWS_OUT ............... 100
EST_OPER_COST ............ 0.02
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. full scan of index
CAT.SCH.PSX2(CAT.SCH.PARTSUPP)
  scan_direction ......... reverse
  object_type ............ CAT.SCH.PARTSUPP
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock cursor
  access_mode ............ not specified, defaulted to read committed
  columns_retrieved ...... 6
  key_columns ............ CAT.SCH.PSX2.PS_PARTKEY,
CAT.SCH.PSX2.PS_SUPPKEY,
                             CAT.SCH.PSX2.PS_SUPPLYCOST,
                             CAT.SCH.PSX2.PS_AVAILQTY,
CAT.SCH.PSX2.PS_PARTKEY,
                             CAT.SCH.PSX2.PS_SUPPKEY
  begin_key .............. (CAT.SCH.PSX2.PS_PARTKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPLYCOST = <max>),
                             (CAT.SCH.PSX2.PS_AVAILQTY = <max>),
                             (CAT.SCH.PSX2.PS_PARTKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <max>)
  end_key ................ (CAT.SCH.PSX2.PS_PARTKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPLYCOST = <min>),
                             (CAT.SCH.PSX2.PS_AVAILQTY = <min>),
                             (CAT.SCH.PSX2.PS_PARTKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <min>)

--- SQL operation complete.
```

- Use `OPTIONS 'm'`:

```
>>explain options 'm' xx;
```

The following output is displayed:

```
MODULE_NAME
STATEMENT_NAME                                                      PLAN_ID
SEQ_NUM      OPERATOR                        LEFT_CHILD_SEQ_NUM
RIGHT_CHILD_SEQ_NUM  TNAME
CARDINALITY OPERATOR_COST TOTAL_COST DETAIL_COST
DESCRIPTION
------------------------------------------------------------  ------------
----------------------------------------------------  --------------------  --
---------  ------------------------------  ------------------  ------------
--------  ------------------------------------------------------------  --
-------------  --------------  --------------  --------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
---------------------------  -------------------

?                                     XX
212122794829778687   1          INDEX_SCAN                 ?
?                  CAT.SCH.PARTSUPP
1.0000000E+002   2.0646531E-002   2.0646531E-002   CPU_TIME: 5.56943e-005
IO_TIME:0.0206465MSG_TIME:0IDLETIME:0PROBES:1
fragment_id: 2 parent_frag: 0 fragment_type: dp2  olt_optimization: not
used olt_opt_lean: not used scan_type: full scan of index
CAT.SCH.PSX2(CAT.SCH.PARTSUPP) scan_direction: reverse object_type:
CAT.SCH.PARTSUPP key_type: simple lock_mode: not specified, defaulted to
lock cursor access_mode: not specified, defaulted to read committed
columns_retrieved: 6  key_columns: CAT.SCH.PSX2.PS_PARTKEY,
CAT.SCH.PSX2.PS_SUPPKEY, CAT.SCH.PSX2.PS_SUPPLYCOST,
CAT.SCH.PSX2.PS_AVAILQTY, CAT.SCH.PSX2.PS_PARTKEY, CAT.SCH.PSX2.PS_SUPPKEY
begin_key: (CAT.SCH.PSX2.PS_PARTKEY = <max>), (CAT.SCH.PSX2.PS_SUPPKEY =
<max>), (CAT.SCH.PSX2.PS_SUPPLYCOST = <max>), (CAT.SCH.PSX2.PS_AVAILQTY =
<max>), (CAT.SCH.PSX2.PS_PARTKEY = <max>), (CAT.SCH.PSX2.PS_SUPPKEY =
<max>) end_key: (CAT.SCH.PSX2.PS_PARTKEY = <min>),
(CAT.SCH.PSX2.PS_SUPPKEY = <min>), (CAT.SCH.PSX2.PS_SUPPLYCOST = <min>),
(CAT.SCH.PSX2.PS_AVAILQTY = <min>), (CAT.SCH.PSX2.PS_PARTKEY = <min>),
(CAT.SCH.PSX2.PS_SUPPKEY = <min>)
?                                     XX
212122794829778687   2          FIRSTN                     1
?
1.0000000E+000   0.0000000E+000   0.0000000E+000   OPERATOR_COST: 0
ROLLUP_COST:0
fragment_id: 2 parent_frag: 0 fragment_type: dp2
?                                     XX
212122794829778687   3         SHORTCUT_SCALAR_AGGR         2
?
1.0000000E+000   3.3796350E-006   1.6517225E-002   CPU_TIME: 4.79351e-005
IO_TIME:0.0165172MSG_TIME:0IDLETIME:0PROBES:1
fragment_id: 2 parent_frag: 0 fragment_type: dp2  aggregates:
max(CAT.SCH.PSX2.PS_PARTKEY)
```

```
?                                    XX
212122794829778687   4         PARTITION_ACCESS            3
?
1.0000000E+000   7.5787946E-004   1.6517225E-002   CPU_TIME: 0.000805815
IO_TIME:0.0165172 MSG_TIME:0 IDLETIME:0 PROBES:1
fragment_id: 2 parent_frag: 0 fragment_type: dp2  buffer_size: 31000
record_length: 8  space_usage: 12:8:8:32  eid_space_computation: on
?                                    XX
212122794829778687   5         FILE_SCAN_UNIQUE            ?
?                  CAT.SCH.PART
1.0000000E+000   2.0646531E-002   2.0646531E-002   CPU_TIME: 3.18424e-005
IO_TIME:0.0206465 MSG_TIME:0 IDLETIME:0 PROBES:1
fragment_id: 3 parent_frag: 0 fragment_type: dp2  olt_optimization: not
used olt_opt_lean: not used scan_type: unique access of table CAT.SCH.PART
object_type: CAT.SCH.PART key_type: simple lock_mode: not specified,
defaulted to lock cursor access_mode: not specified, defaulted to read
committed columns_retrieved: 9 fast_replydata_move: used  key_columns:
P_PARTKEY key: (P_PARTKEY = max(CAT.SCH.PSX2.PS_PARTKEY))
?                                    XX
212122794829778687   6         PARTITION_ACCESS            5
?
1.0000000E+000   7.6004536E-004   2.0646531E-002   CPU_TIME: 0.000791888
IO_TIME:0.0206465 MSG_TIME:0 IDLETIME:0 PROBES:1
fragment_id: 3 parent_frag: 0 fragment_type: dp2  buffer_size: 31000
record_length: 177  space_usage: 13:8:8:36  eid_space_computation: on
?                                    XX
212122794829778687   7         NESTED_JOIN                 4
6
1.0000000E+000   4.1742293E-007   3.7163756E-002   CPU_TIME: 0.00159812
IO_TIME:0.0371638 MSG_TIME:0 IDLETIME:0 PROBES:1
fragment_id: 0 parent_frag: (none) fragment_type: master join_type: inner
join_method: nested
?                                    XX
212122794829778687   8         ROOT                        7
?
1.0000000E+000   1.9600035E-001   2.3316375E-001   CPU_TIME: 0.00159847
IO_TIME:0.0371638 MSG_TIME:0 IDLETIME:0.196 PROBES:1
fragment_id: 0 parent_frag: (none) fragment_type: master  statement_index:
0 statement: select * from part where p_partkey = (select max(ps_partkey)
from partsupp); xn_access_mode: read_only plan_version: 2400 SCHEMA:
CAT.SCH  select_list: CAT.SCH.PART.P_PARTKEY, CAT.SCH.PART.P_NAME,
CAT.SCH.PART.P_MFGR, CAT.SCH.PART.P_BRAND, CAT.SCH.PART.P_TYPE,
CAT.SCH.PART.P_SIZE, CAT.SCH.PART.P_CONTAINER, CAT.SCH.PART.P_RETAILPRICE,
CAT.SCH.PART.P_COMMENT

--- SQL operation complete.
```

- Use the EXPLAIN statement without the output options:

  ```
  >>explain xx;
  ```

  The following output is displayed:

```
------------------------------------------------------------------ PLAN
SUMMARY
MODULE_NAME .............. DYNAMICALLY COMPILED
STATEMENT_NAME ........... XX
PLAN_ID .................. 212122794829778687
ROWS_OUT ................. 1
EST_TOTAL_COST ........... 0.23
STATEMENT ................ select *
                           from part
                           where p_partkey = (select max(ps_partkey) from
                             partsupp);


------------------------------------------------------------------ NODE
LISTING
ROOT ======================================  SEQ_NO 8      ONLY CHILD 7
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.2
EST_TOTAL_COST ........... 0.23
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  statement_index ........ 0
  xn_access_mode ......... read_only
  plan_version ....... 2,400
  SCHEMA ................. CAT.SCH
  select_list ............ CAT.SCH.PART.P_PARTKEY, CAT.SCH.PART.P_NAME,
                             CAT.SCH.PART.P_MFGR, CAT.SCH.PART.P_BRAND,
                             CAT.SCH.PART.P_TYPE, CAT.SCH.PART.P_SIZE,
                             CAT.SCH.PART.P_CONTAINER,
                             CAT.SCH.PART.P_RETAILPRICE,
CAT.SCH.PART.P_COMMENT


NESTED_JOIN ===============================  SEQ_NO 7      CHILDREN 4, 6
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.01
EST_TOTAL_COST ........... 0.04
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner
  join_method ............ nested


PARTITION_ACCESS ==========================  SEQ_NO 6      ONLY CHILD 5
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.01
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  buffer_size ....... 31,000
  record_length ........ 177
  space_usage ............ 13:8:8:36
  eid_space_computation    on
```

```
FILE_SCAN_UNIQUE =========================  SEQ_NO 5      NO CHILDREN
TABLE_NAME ............... CAT.SCH.PART
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.02
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. unique access of table CAT.SCH.PART
  object_type ............ CAT.SCH.PART
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock cursor
  access_mode ............ not specified, defaulted to read committed
  columns_retrieved ...... 9
  fast_replydata_move .... used
  key_columns ............ P_PARTKEY
  key .................... (P_PARTKEY = max(CAT.SCH.PSX2.PS_PARTKEY))


PARTITION_ACCESS =========================  SEQ_NO 4      ONLY CHILD 3
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.01
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  buffer_size ....... 31,000
  record_length .......... 8
  space_usage ............ 12:8:8:32
  eid_space_computation    on


SHORTCUT_SCALAR_AGGR =====================  SEQ_NO 3      ONLY CHILD 2
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0.01
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  aggregates ............. max(CAT.SCH.PSX2.PS_PARTKEY)


FIRSTN ===================================  SEQ_NO 2      ONLY CHILD 1
REQUESTS_IN .............. (not found)
ROWS_OUT ................. 1
EST_OPER_COST ............ 0
EST_TOTAL_COST ........... 0
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
```

```
INDEX_SCAN ================================  SEQ_NO 1      NO CHILDREN
TABLE_NAME ............... CAT.SCH.PARTSUPP
REQUESTS_IN .............. 1
ROWS_OUT ............... 100
EST_OPER_COST ............ 0.02
EST_TOTAL_COST ........... 0.02
DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. full scan of index
CAT.SCH.PSX2(CAT.SCH.PARTSUPP)
  scan_direction ......... reverse
  object_type ............ CAT.SCH.PARTSUPP
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock cursor
  access_mode ............ not specified, defaulted to read committed
  columns_retrieved ...... 6
  key_columns ............ CAT.SCH.PSX2.PS_PARTKEY,
CAT.SCH.PSX2.PS_SUPPKEY,
                             CAT.SCH.PSX2.PS_SUPPLYCOST,
                             CAT.SCH.PSX2.PS_AVAILQTY,
CAT.SCH.PSX2.PS_PARTKEY,
                             CAT.SCH.PSX2.PS_SUPPKEY
  begin_key .............. (CAT.SCH.PSX2.PS_PARTKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPLYCOST = <max>),
                             (CAT.SCH.PSX2.PS_AVAILQTY = <max>),
                             (CAT.SCH.PSX2.PS_PARTKEY = <max>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <max>)
  end_key ................ (CAT.SCH.PSX2.PS_PARTKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPLYCOST = <min>),
                             (CAT.SCH.PSX2.PS_AVAILQTY = <min>),
                             (CAT.SCH.PSX2.PS_PARTKEY = <min>),
                             (CAT.SCH.PSX2.PS_SUPPKEY = <min>)

--- SQL operation complete.
>>
```

- This is an example of `insert…select` with an IDENTITY column:

```
create table tab1(a largeint generated always as identity, b
int);


--- SQL operation complete.

prepare xx from insert into tab1 values(DEFAULT, 1);


--- SQL command prepared.

explain options 'f' xx;
```

```
LC   RC   OP   OPERATOR              OPT       DESCRIPTION           CARD
---- ---- ---- -------------------  --------  --------------  ---------


9    .    10   root                             r                     1.00E+000
6    8    9    tuple_flow                                             1.00E+000
7    .    8    partition_access                                       1.00E+000
.    .    7    insert                           TAB1                  1.00E+000
1    5    6    nextvaluefor                                           1.00E+000
4    .    5    esp_access                                             1.00E+000
3    .    4    sequencegenerator                                      1.00E+000
2    .    3    partition_access                                       1.00E+002
.    .    2    subset_update                  "@@INTERNAL_SG_19505   1.00E+002
.    .    1    values                                                 1.00E+000

--- SQL operation complete.
>>
```

FASTCOPY Utility

Considerations for FASTCOPY
Examples of FASTCOPY

FASTCOPY is a syntax-based utility that can be executed through MXCI and from a program using dynamic SQL. The RECOVER support is available for the FASTCOPY utility. FASTCOPY is not available as an embedded SQL. For more information about the FASTCOPY utility, see the *SQL/MX Installation and Management Guide*.

Note. The FASTCOPY utility is available only on systems running J06.08 and later J-series RVUs and H06.19 and later H-series RVUs.

The two forms of FASTCOPY utility are:

- FASTCOPY TABLE Command
- FASTCOPY INDEX Command

FASTCOPY TABLE Command

The FASTCOPY TABLE command copies all the rows from one table to the existing equivalent table.

```
FASTCOPY TABLE source-table [TO] target-table
[ index-clause ]

index-clause is
  INDEXES { EXPLICIT | IMPLICIT }
```

source-table

is the ANSI name of the table where rows are copied from.

target-table

is the ANSI name of the table that is being copied into. The target table must exist prior to the operation and must be equivalent with the source table. The existing rows in the target table will be removed before the copying begins.

index-clause

specifies how FASTCOPY TABLE will treat indexes. The *index-clause* has these values:

EXPLICIT	Indexes on the target table will be explicitly copied. For each matching pair of source and target indexes, the FASTCOPY INDEX Command can be used to explicitly perform the copy concurrently with the copy of the table, if required. Each target index must have a matching source index.
IMPLICIT	All indexes on the target table are maintained automatically as a part of the FASTCOPY utility.

The default is INDEXES IMPLICIT.

FASTCOPY INDEX Command

The FASTCOPY INDEX command copies all rows from one index to the existing equivalent index.

```
FASTCOPY INDEX source-index [TO] target-index
```

source-index

is the ANSI name of the index that is being copied from.

target-index

is the ANSI name of the index that is being copied into. The target index must exist prior to the operation.

Considerations for FASTCOPY

- The FASTCOPY TABLE command copies column values by the column order—the first source table column is copied to the first target table column, the second source table column is copied to the second target table column, and so forth.
- The FASTCOPY command preserves the SYSKEY column values from the source table. The copy-by-column-order strategy applies to SYSKEY columns as well.
- As part of initialization, the FASTCOPY TABLE command removes the existing rows from the target table and all its indexes.
- If the fastcopy operation is incomplete because of any reason, use the RECOVER utility before reattempting a FASTCOPY command on the same table.
- For a table with an IDENTITY column at the source, the target, or both, the FASTCOPY utility copies the source base table data to the target base table, according to existing FASTCOPY logic for base tables. In addition, FASTCOPY

updates the `CURRENT_VALUE` in the target SG Table to match the copied base table data, according to the rules described in Table 2-7.

Table 2-7. Rules for copying SG Table data

Target column	Source column	Expected behavior
IDENTITY ALWAYS	NO IDENTITY COLUMN	FASTCOPY issues an error. If the target column has IDENTITY ALWAYS, the source column must also have IDENTITY ALWAYS for the DEFAULT specification type.
IDENTITY ALWAYS	NO IDENTITY COLUMN	FASTCOPY issues an error. If the target column has IDENTITY ALWAYS, the source column must also have IDENTITY ALWAYS for the DEFAULT specifica-ion type.

Table 2-7. Rules for copying SG Table data

IDENTITY BY DEFAULT	IDENTITY BY DEFAULT	FASTCOPY unconditionally copies the base table data. FASTCOPY copies the CURRENT_VALUE of the SG TABLE only if the SG Attributes and position of the IDENTITY column in the source and target table match. Otherwise, the CURRENT_VALUE of the target SG TABLE will not be updated. To set the CURRENT_VALUE of the target SG TABLE, the user can do ALTER TABLE ALTER COLUMN RECALIBRATE.
IDENTITY BY DEFAULT	IDENTITY ALWAYS	FASTCOPY unconditionally copies the base table data. FASTCOPY copies the CURRENT_VALUE of the SG_TABLE only if the SG Attributes and position of the IDENTITY column in the source-and target table match. Otherwise, the CURRENT_VALUE of the target SG Table will not be updated. To set the CURRENT_VALUE of the target SG Table, the user can do ALTER TABLE ALTER COLUMN RECALIBRATE.
IDENTITY column	IDENTITY • BY DEFAULT • ALWAYS	FASTCOPY unconditionally copies the base table data. Target SG Table does not exist since the target table does not contain an IDENTITY column.

Equivalence Requirements

- The target table and the source table must be sufficiently equivalent so that the following command can be executed successfully:

  ```
  insert into <target table> (*) select * from <source table>;
  ```

  The two tables must have the same number of columns, and the columns with the same ordinal position must have compatible data types. Also, either both tables must have a SYSKEY column or none of the tables may have a SYSKEY column.

  If the target table has indexes that are online, the clustering key for the target table must be equivalent to the clustering key for the source table. The following details must be identical between the clustering keys:

  - number of columns
  - ordinal position of each clustering key column within its table

Matching Indexes

- If the target table has indexes that are online, the FASTCOPY TABLE command determines the source and target indexes that match, if any. A target index matches a source index if their index specifications are equivalent. That is, if all of the following conditions are satisfied:

  - the index key for the target index must be equivalent to the index key for the source index—the following details must be identical between the index keys:
    - number of columns
    - ordinal position of each index key column within its table
  - the source index is not offline

  Each target index that is not offline must have a matching source index. The FASTCOPY utility does not consider target indexes that are offline.

  If the INDEXES EXPLICIT option is set, you must specify the matching source and target indexes on the related FASTCOPY INDEX Command.

  - If the INDEXES IMPLICIT option is set, the system will select a matching source index for each involved target index.

Availability of Source and Target Tables

- While executing the FASTCOPY TABLE command, the source table and its definition are available in read-only mode. That is, you will not be able to perform the following operations on the source table:

  - insert, update, and delete DML operations
  - DDL operations on the source table or its indexes

- utility operations that modify the data or definition of the source table or its indexes

Data in the target table are not available while executing the FASTCOPY TABLE or FASTCOPY INDEX command. Data in the target table can be accessed only after the FASTCOPY operation completes successfully. The definition of the target table(DDL) are available for read-only operations.

Recovery

- The RECOVER INDEX command can be used only when explicit copying of index rows is involved. That is, recovery of a failed FASTCOPY TABLE...INDEXES IMPLICIT must use the RECOVER TABLE command.
- When the CANCEL option is set, the entire fastcopy operation is canceled. RECOVER...CANCEL applies to entire fastcopy operation. It restores the source and target tables to the original state with an exception that the purge on the target table cannot be rolled back. If you perform RECOVER…CANCEL on an actively running FASTCOPY command, it will return an error 20212.
- When the RESUME option is set, it affects only that target object which is mentioned in the RECOVER command. The target object will be set to its initial state and then the corresponding FASTCOPY command will be implicitly executed.

DDL Locks

To support the fastcopy operation, the following types of DDL_LOCKS metadata rows are used:

- DDL_LOCKS for source objects

  For source objects, the DDL_LOCKS row displays an operation value Fastcopy Source (FS). The status indicates how indexes are treated. The status values are listed below:

Value	Description
0	INDEXES IMPLICIT is specified to FASTCOPY TABLE.
1	INDEXES EXPLICIT is specified to FASTCOPY TABLE or the originating command is FASTCOPY INDEX.

  For source objects, only DDL_LOCKS rows for tables have an associated TEXT row that points to the target table. Also, for source tables there are additional TEXT rows (with `object_sub_id` 19) that contain information about all source table partitions.

- DDL_LOCKS for target objects

  For target objects, the DDL_LOCKS row displays an operation value Fastcopy Target (FT). The status indicates the actual progress of the FASTCOPY of that particular target object. The status values are listed below:

Value	Description
1	A fastcopy operation is started. Audit is turned off and all target partitions (table and indexes) are marked "corrupt". A target object with this DDL lock state is ready for a continuation FASTCOPY command.
2	The object is ready for copy. The actual row copying is about to start for the target object. A target object with this DDL lock state or any subsequent state is processed by a FASTCOPY command, and therefore, is not processed by another FASTCOPY command.
3	Copy is in progress. Actual row copying is ongoing. The target object is not marked "corrupt" when the object is in this state.
4	Copy is completed. The target object is marked "corrupt".
5	The operation is completed. This is an intermediate state, which indicates that all involved objects (table, indexes) are successfully copied and the entire fastcopy operation is about to complete.
6	Cancel the operation after copy. The fastcopy operation is canceled. However, the FASTCOPY command copies rows for the affected object. This state signals the FASTCOPY command that the operation is canceled.
7	Cancel is in progress. The fastcopy operation is canceled.
8	Cancel is completed. This is an intermediate state. It indicates that the fastcopy operation is canceled for all involved objects (table, indexes).

  For target objects, DDL_LOCKS rows for indexes might not initially have an associated TEXT row, when INDEXES EXPLICIT is used. The TEXT row will be created as part of FASTCOPY INDEX.

Examples of FASTCOPY

- Consider a source table ST with indexes SI1 and SI2 and the target table TT with index TI1. Run the following command:

  ```
  FASTCOPY TABLE ST TO TT INDEXES IMPLICIT;
  ```

  The command copies all the rows from the source table to the target table and automatically maintains index TI1 as part of that copying.

  When you specify the *index-clause* as EXPLICIT:

  ```
  FASTCOPY TABLE ST TO TT INDEXES EXPLICIT;
  ```

  The command copies all the rows from the source table to the target table. An explicit FASTCOPY of source index SI1 to target index TI1 is required to complete the fastcopy operation. Because the source index SI2 has no equivalent target index, it does not participate in the fastcopy operation.

  You can also start the explicit fastcopy operation with the FASTCOPY INDEX command (not necessarily with the FASTCOPY TABLE command) followed by the required number of additional FASTCOPY INDEX commands and one FASTCOPY TABLE...INDEXES EXPLICIT command, in any order.

  To copy all the rows from source index SI1 to target index TI1, run the following command:

  ```
  FASTCOPY INDEX SI1 to TI1;
  ```

  The base table TT can be accessed only after completing the fastcopy operation (copying of indexes SI1 to TI1). If you try to access the table without completing the fastcopy operation (by copying the index explicitly), it returns the following error:

  ```
  ERROR[8580] No partitions of table could be accessed
  ```

GET ALL SECURITY_ADMINS Statement

The GET ALL SECURITY_ADMINS statement lists the users in the Security Administrators Group.

GET ALL SECURITY_ADMINS is an SQL/MX extension.

```
GET ALL SECURITY_ADMINS
```

Considerations for GET ALL SECURITY_ADMINS

Authorization Requirements

Any Guardian user may execute the GET ALL SECURITY_ADMINS statement.

Metadata Version Requirements

The GET ALL SECURITY_ADMINS statement requires system metadata version 3100 or greater. If the statement is executed with lower versions of the system metadata, a SQL error is generated indicating that the SYSTEM_SECURITY_SCHEMA does not exist.

Invalid Security Administrator User IDs

If a user ID that does not exist in the system USERID file is encountered while processing the GET ALL SECURITY_ADMINS statement, it will be mapped to its numeric representation. This situation can occur if an existing user ID is designated as a security administrator and then later removed from the system USERID file without revoking the security administrator designation.

Examples of GET ALL SECURITY_ADMINS

- The following example lists the security administrators:

```
GET ALL SECURITY_ADMINS;

Security Administrators

=======================

SECADMIN.USER1

SECADMIN.USER2

--- SQL operation complete.
```

- The following example lists the security administrators when one of the security administrators is no longer in the system USERID file:

```
GET ALL SECURITY_ADMINS;

Security Administrators

=======================

SECADMIN.USER1

4,85

--- SQL operation complete.
```

- The following example lists the security administrators when the security administrators group is empty:

```
GET ALL SECURITY_ADMINS;

--- SQL operation complete.
```

GIVE CATALOG Statement

Considerations for GIVE CATALOG
Example of GIVE CATALOG

The GIVE CATALOG statement transfers the ownership of catalog from one Guardian user to another.

```
GIVE CATALOG catalog TO authid
```

catalog

is an SQL identifier that specifies the name of a catalog.

authid

specifies an authorization ID to whom the ownership is transferred. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name, enclosed in double quotes. A Guardian user number (for example, “255,255”) is not allowed. *authid* is not case-sensitive.

Considerations for GIVE CATALOG

Object Feature Version of the objects under the catalog is not changed as part of this operation.

Authorization and Availability Requirements

The GIVE CATALOG operation can be performed by a catalog owner or security administrator or the Super ID (if Super ID is part of the Security Administrator’s group or if no Security Administrator’s group exists). The ownership of a catalog cannot be transferred to PUBLIC. The ownership of a catalog cannot be transferred to any member of the Security Administrator’s group unless the GRANTOR is the owner of the catalog. The Catalog ownership change does not invalidate the existing opens on the objects in the schemas residing in the catalog. After successful completion of the GIVE CATALOG operation, only the new owner can drop a catalog.

Example of GIVE CATALOG

- This example transfers the catalog ownership to “SQL.USER1”.

```
>>give catalog usercat to "sql.user1";
--- SQL operation complete.
```

GIVE Object Statement

Considerations for GIVE Object
Examples of GIVE Object

The GIVE object statement transfers the ownership of the object from one Guardian user to another.

```
GIVE object-type object TO authid
```

object-type

specifies the type of the object whose ownership will be changed. The object type can be one of the following:
`{TABLE | TRIGGER | VIEW | PROCEDURE | SEQUENCE}`

object

specifies the name of the object whose ownership will be changed. The object name can either be fully qualified or partially qualified. If the name is not fully qualified, the object name is resolved based on default catalog and schema names.

authid

specifies an authorization ID to whom the ownership is transferred. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name, enclosed in double quotes. A Guardian user number (for example, “255,255”) is not allowed. *authid* is not case-sensitive.

Considerations for GIVE Object

The GIVE TABLE operation will change the ownership of associated indexes, trigger temporary table and constraints to the new owner.

The GIVE VIEW operation will not change the ownership of underlying tables.

Object Feature Version (OFV) of the object is increased to 3100, if the schema owner and object owner are different after the completion of the operation. However, after completion of the operation, if the schema owner and the object owner happen to be the same again, then the OFV is recomputed and set.

Authorization and Availability Requirements

The GIVE operation on a Table, View, Procedure, Trigger and Sequence Generator can be performed by schema owner or object owner or security administrator or the Super ID (if Super ID is part of the Security Administrator’s group or if no Security Administrator’s group exists). The ownership of an object cannot be transferred to PUBLIC. The ownership of an object cannot be transferred to any member of the Security Administrator’s group, unless the GRANTOR is the object owner or the schema owner. After successful completion of this operation, all the existing opens of the target object are invalidated. In case of VIEW ownership change, the existing opens on the underlying tables are also invalidated.

Examples of GIVE Object

- This example transfers the table ownership to “SQL.USER2”.

```
>>give table usertable to "sql.user2";
--- SQL operation complete.
```

- This example transfers view ownership to “SQL.USER5”.

```
>>give view userview to "sql.user5";
--- SQL operation complete.
```

- This example transfers procedure ownership to “SQL.USER3”.

```
>>give procedure userroutine to "sql.user3";
--- SQL operation complete.
```

- This example transfers trigger ownership to “SQL.USER1”:

```
>>give trigger usertrigger to "sql.user1";
--- SQL operation complete.
```

- This example transfers sequence generator ownership to “SQL.USER1”:

```
>>give sequence myseq to "sql.user1";
--- SQL operation complete.
```

GIVE SCHEMA Operation

Considerations for GIVE SCHEMA
Examples of GIVE SCHEMA

The GIVE SCHEMA operation transfers the ownership of the schema and optionally, those objects in the schema owned by the schema owner from one user to another.

```
GIVE SCHEMA schema TO authid [CASCADE]
```

schema

specifies the schema on which the GIVE operation is performed.

authid

specifies an authorization ID to whom the ownership is transferred. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name, enclosed in double quotes. A Guardian user number (for example, “255,255”) is not allowed. *authid* is not case-sensitive.

CASCADE

specifies whether the ownership changes are cascaded to the objects owned by the schema owner within the schema.

Considerations for GIVE SCHEMA

The ownership of user metadata tables in the schema is transferred to the new owner irrespective of the CASCADE option.

If the schema owner and object owner are different after the completion of the operation, the Object Feature Version (OFV) of the objects within the *schema* is at least 3100. However, if after completion of the operation, the schema owner and object owner are the same again, then the OFV is recomputed and set.

Authorization and Availability Requirements

The GIVE SCHEMA operation can be performed by the schema owner or security administrator or a Super ID (if Super ID is part of the Security Administrator’s group or if no Security Administrator’s group exists). After successful completion of this operation, all the existing opens on the underlying objects are invalidated.

The ownership of a schema cannot be transferred to PUBLIC. The ownership of a schema cannot be transferred to any member of the Security Administrator’s group, unless the GRANTOR is the owner of the schema. The ownership of a schema cannot be transferred by any member of Security Administrator’s group to themselves. If Super ID is not part of Security Administrator’s Group, it can transfer ownership of schema to itself.

Examples of GIVE SCHEMA

- This example transfers the schema ownership to “SQL.USER5”:

```
>>give schema userschema1 to "sql.user5";
--- SQL operation complete.
```

- This example transfers the schema and the objects within the schema ownership to “SQL.USER5” :

```
>>give schema userschema1 to "sql.user5" cascade;
--- SQL operation complete.
```

DDL Locks

During GIVE SCHEMA operation, DDL lock is held to prevent any other concurrent DDL or utility operation being performed on the schema. The lock is released on successful completion of the GIVE SCHEMA operation.

If the GIVE SCHEMA operation fails unexpectedly, you must run the RECOVER SCHEMA operation on that schema to recover from the failed GIVE SCHEMA operation. After successful recovery, the DDL lock will be released.

GRANT Statement

Considerations for GRANT
Examples of GRANT

The GRANT statement grants access privileges for an SQL/MX table, view, sequence generator, or stored procedure to specified users. For more information, see GRANT EXECUTE Statement on page 2-246 .

```
GRANT {privilege [,privilege]... | ALL [PRIVILEGES]}
   ON [TABLE | SEQUENCE] object
   TO {grantee [,grantee ]...}
   [WITH GRANT OPTION]
   [BY authid-grantor]

grantee is:
   authid | PUBLIC

privilege is:
     SELECT
   | DELETE
   | INSERT
   | UPDATE [(column [,column]...)]
   | REFERENCES [(column [,column]...)]
   | USAGE
```

Syntax Description of GRANT

`privilege [,privilege]... | ALL [PRIVILEGES]`

specifies the privileges to grant. You can specify each of these privileges for a table or a view. See also the GRANT EXECUTE Statement on page 2-246.

SELECT	Can use SELECT statement.
DELETE	Can use DELETE statement.
INSERT	Can use INSERT statement.
UPDATE	Can use UPDATE statement.
REFERENCES	Can create constraints that reference the object.
USAGE	Can use the pseudocolumns, CURRVAL and NEXTVAL to access sequence generator values.
ALL PRIVILEGES	Can have all privileges that apply to the object type. When ALL is specified, the object can be a table, view, sequence generator, or stored procedure. When the object is a stored procedure and ALL is specified, only EXECUTE permission is applied. When the object is a sequence generator and ALL is specified, only USAGE privilege is applied.

(*column* [,*column*]...)

names the columns of the object to which the UPDATE or REFERENCES privileges apply. If you specify UPDATE or REFERENCES without column names, the privileges apply to all columns of the table or view.

`ON [TABLE | SEQUENCE] object`

specifies a table, view, sequence generator or stored procedure on which to grant privileges. When the object is a stored procedure, the only privileges you can specify are ALL PRIVILEGES or EXECUTE. See GRANT EXECUTE Statement on page 2-246.

`TO {authid [,authid ]... | PUBLIC}`

specifies one or more users to whom you grant privileges.

authid specifies an authorization ID to whom you grant privileges. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name, enclosed in double quotes. A Guardian user number (for example, “255,255”) is not allowed. *authid* is not case-sensitive.

SQL:1999 specifies two special authorization IDs: PUBLIC and SYSTEM.

- PUBLIC specifies all present and future authorization IDs.

- SYSTEM specifies the implicit grantor of privileges to the creators of objects.

  You cannot specify SYSTEM as an *authid* in a GRANT statement.

`WITH GRANT OPTION`

specifies that users of the authorization IDs to whom privileges are granted have the right to grant the same privileges to other authorization IDs.

`BY` *`authid-grantor`*

specifies the authorization ID *authid-grantor* on whose behalf the grant operation is performed. Only the Super ID can use the `BY` clause unless the Security Administrators Group is not empty and the Super ID has not been designated as a Security Administrator in which case the Super ID is denied the use of this feature. The effect of using the `BY` clause is the same as if the *authid-grantor* were to issue the GRANT directly (without using the `BY` clause). *authid-grantor* cannot be SYSTEM. If the Security Administrator's Group is empty, then *authid-grantor* must be a valid authorization ID and hold the privilege(s) being granted WITH GRANT OPTION. However, if the Super ID is designated as a Security Administrator, it will have Super `GRANT BY` capabilities in which *authid-grantor* may be any valid authorization ID.

Considerations for GRANT

Authorization Requirements

Unless you are a Security Administrator or the Super ID, to grant a privilege on an object, you must have both that privilege and the right to grant that privilege. That is, the privilege must have been issued to you WITH GRANT OPTION and not revoked. If you lack authority to grant one or more of the specific privileges, the system returns a warning (and does perform the grant of any of the specified privileges that you do have authority to grant). If you have none of the specified privileges WITH GRANT OPTION, the system returns an error.

If you are a Security Administrator, then you are exempt from the above restriction and may grant a privilege without having the privilege. However, such grants may not be made to PUBLIC or a Security Administrator or using WITH GRANT OPTION. Security Administrators may hold a derived WGO privilege, in which case they may grant that privilege like any other user (including to PUBLIC and using WITH GRANT OPTION). This latter type of grant is included in the hierarchy of owner-derived grants.

If you are the Super ID, then your grant privileges depend on the Security Administrator's Group. If the Security Administrator's Group is empty, then you may grant any privilege on any object. Such grants behave like a `GRANT BY` *`authid-grantor`* where the *`authid-grantor`* is the object owner.

If the Super ID is designated as a Security Administrator, then the Super ID has the same privileges as any other Security Administrator plus the ability to execute `GRANT BY` *`authid-grantor`*. If `BY` *`authid-grantor`* is omitted, then the implied grantor is the Super ID instead of the object owner.

If the Security Administrator's Group is not empty and the Super ID is not designated as a Security Administrator, the Super ID will have the same restrictions as any ordinary user with respect to the GRANT statement.

To grant the USAGE privilege on a sequence generator, you must have the USAGE privilege and the right to grant this privilege. The owner, or creator of the sequence generator, Security Administrator and the Super ID automatically have USAGE and WITH GRANT OPTION privileges on a sequence generator. All other users must be granted both the USAGE and WITH GRANT OPTION privileges to grant USAGE privilege to other users. If an unauthorized user attempts to grant the USAGE privilege, an error is returned. If the Super ID issues a GRANT USAGE statement using the BY *`authid-grantor`* clause, the *`authid-grantor`* must have the right to grant the USAGE privilege.

Security Considerations

NonStop SQL/MX translates each authorization ID you specify into a 32-bit integer value, and then stores the number in the system metadata tables. The stored identification number, not the characters of the authorization ID, is used to identify the user who holds privileges on the specified objects.

Privileges on Views

Granting a privilege on a view does not grant that privilege to the corresponding column of the underlying table.

Privileges on Stored Procedures

You can also manage security on a stored procedure by using the GRANT EXECUTE and REVOKE EXECUTE statements. See GRANT EXECUTE Statement on page 2-246 and REVOKE EXECUTE Statement on page 2-323.

Examples of GRANT

- This example grants SELECT and DELETE privileges on a table, in addition to the privilege of granting SELECT and DELETE privileges to others:

```
GRANT SELECT, DELETE ON TABLE sales.odetail
  TO "sql.user1", "sql.user2" WITH GRANT OPTION;
```

- This example grants UPDATE privileges on the named columns to PUBLIC:

```
GRANT UPDATE (start_date, ship_timestamp)
  ON TABLE persnl.project TO PUBLIC;
```

- In this example, the super ID grants SELECT and DELETE privileges on a table on behalf of `sql.user1`:

```
GRANT SELECT, DELETE ON TABLE sales.odetail
  TO "sql.user3" BY "sql.user1";
```

- This example gives USAGE privilege on a sequence generator to `sql.user2`:

```
GRANT USAGE ON myseq to "sql.user2";
```

- This example gives USAGE privilege on a sequence generator to multiple users WITH GRANT OPTION:

```
GRANT USAGE ON myseq to "sql.user3","sql.user5" WITH GRANT
OPTION;
```

GRANT CREATE CATALOG Statement

Considerations for GRANT CREATE CATALOG
Examples for GRANT CREATE CATALOG

The GRANT CREATE CATALOG grants privileges to create a catalog for specified users.

```
GRANT CREATE CATALOG TO {"grantee" [,"grantee"]...}
```

{"*grantee*" [,"*grantee*"]...}

are the recipients of the GRANT privileges.

Considerations for GRANT CREATE CATALOG

Only authorized users are allowed to create catalog after the successful completion of the first GRANT CREATE CATALOG statement.

Authorization and Availability Requirements

A security administrator or the Super ID (if Super ID is part of the Security Administrator’s group or if no Security Administrator’s group exists) can grant the privilege to create a catalog.

- CREATE CATALOG privilege cannot be granted to PUBLIC.
- CREATE CATALOG privilege cannot be granted to any member of the Security Administrator’s group.

Examples for GRANT CREATE CATALOG

- This example grants users, “SQL.USER1”, “SQL.USER2”, and “SQL.USER3” with the privilege to create a catalog:

```
GRANT CREATE CATALOG TO "SQL.USER1";
GRANT CREATE CATALOG TO "SQL.USER2", "SQL.USER3";
```

GRANT CREATE SCHEMA Statement

Considerations for GRANT CREATE SCHEMA
Example for GRANT CREATE SCHEMA

The GRANT CREATE SCHEMA grants privileges to create a schema on a specified catalog to specified users.

```
GRANT CREATE SCHEMA ON catalog TO {"grantee" [,"grantee"]...}
```

{"*grantee*" [,"*grantee*"]...}

are the recipients of the GRANT privileges.

catalog

is the name of the catalog, under which the recipients of the GRANT CREATE SCHEMA privilege can create the schema.

Considerations for GRANT CREATE SCHEMA

Only authorized users are allowed to create schema after the successful completion of the first GRANT CREATE SCHEMA statement in the target catalog.

Authorization and Availability Requirements

A security administrator or catalog owner or the Super ID (if Super ID is part of the Security Administrator's group or if no Security Administrator's group exists) can grant the privilege to create a schema.

- CREATE SCHEMA privilege cannot be granted to PUBLIC.
- CREATE SCHEMA privilege cannot be granted to any member of the Security Administrator's group unless the GRANTOR is the owner of the catalog.

Example for GRANT CREATE SCHEMA

- This example grants the users, "SQL.USER1" and "SQL.USER2" with the privilege to create a schema within the catalog:

```
GRANT CREATE SCHEMA ON CAT TO "SQL.USER1", "SQL.USER2";
```

GRANT EXECUTE Statement

Considerations for GRANT EXECUTE
Examples of GRANT EXECUTE

The GRANT EXECUTE statement grants privileges for calling a stored procedure in Java (SPJ) to one or more specified users.

```
GRANT EXECUTE
   ON [PROCEDURE] procedure-ref
   TO {grantee [,grantee ]...}
   [WITH GRANT OPTION]
   [BY authid-grantor]

procedure-ref is:
   [[catalog-name.]schema-name.]procedure-name

grantee is:
   authid | PUBLIC
```

`EXECUTE`

specifies the privilege of calling the stored procedure.

`ON [PROCEDURE] procedure-ref`

specifies the ANSI logical name of a stored procedure on which to grant EXECUTE privilege, of the form:

`[[catalog-name.]schema-name.]procedure-name`

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

`TO {authid [,authid ]... | PUBLIC}`

specifies one or more users to whom you grant EXECUTE privilege.

authid specifies an authorization ID to whom you grant the EXECUTE privilege. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name, enclosed in double quotes. A Guardian user number (for example, “255,255”) is disallowed. *authid* is not case-sensitive.

SQL:1999 specifies two special authorization IDs: PUBLIC and SYSTEM.

- PUBLIC specifies all present and future authorization IDs.
- SYSTEM specifies the implicit grantor of privileges to the creators of stored procedures.

You cannot specify SYSTEM as an *authid* in a GRANT EXECUTE statement.

`WITH GRANT OPTION`

specifies that users of the authorization IDs to whom the EXECUTE privilege is granted have the right to grant EXECUTE privilege to other authorization IDs.

`BY authid-grantor`

specifies the authorization ID *authid-grantor* on whose behalf the grant operation is performed. Only the super ID can use the BY clause. If another user attempts to do so, the system returns an error. The effect of using the BY clause is the same as if the *authid-grantor* were to issue the GRANT EXECUTE statement directly (without using the BY clause).

authid-grantor must be a valid authorization ID and cannot be SYSTEM.

Considerations for GRANT EXECUTE

Authorization and Availability Requirements

To grant EXECUTE privilege on a stored procedure, you must have both that privilege and the right to grant that privilege. The owner, or creator, of the stored procedure and the super ID automatically have EXECUTE and WITH GRANT OPTION privileges on a stored procedure. All other users must be granted both EXECUTE and WITH GRANT

OPTION privileges to grant other users the EXECUTE privilege. If you lack authority to grant the EXECUTE privilege, the system returns an error.

If the super ID issues a GRANT EXECUTE statement using the `BY` *`authid-grantor`* clause, the *`authid-grantor`* must have the right to grant the EXECUTE privilege.

Security Considerations

NonStop SQL/MX translates each authorization ID you specify into a 32-bit integer value and then stores the number in the system metadata tables. The stored identification number, not the characters of the authorization ID, is used to identify the user who holds privileges on the specified objects.

Examples of GRANT EXECUTE

- Suppose that the super ID on behalf of the owner of the SPJ, `'SYSMGT.ANDY'`, grants EXECUTE and WITH GRANT OPTION privileges on ADJUSTSALARY to two other users, `'SYSMGT.BEN'` and `'SYSMGT.JASON'`:

  ```
  GRANT EXECUTE
    ON PROCEDURE samdbcat.persnl.adjustsalary
    TO 'SYSMGT.BEN', 'SYSMGT.JASON'
    WITH GRANT OPTION
    BY 'SYSMGT.ANDY';
  ```

  The users, `'SYSMGT.BEN'` and `'SYSMGT.JASON,'` can then issue EXECUTE and WITH GRANT OPTION privileges to other users on the system. They can also execute CALL statements.

- The user `'SYSMGT.BEN'` grants EXECUTE and WITH GRANT OPTION privileges on `spj1` to user `'HR.BETTY'`:

  ```
  GRANT EXECUTE
    ON PROCEDURE samdbcat.persnl.spj1
    TO 'HR.BETTY'
    WITH GRANT OPTION;
  ```

- The user `'HR.BETTY'` grants EXECUTE privilege on `spj1` to some users in the HR group:

  ```
  GRANT EXECUTE
    ON PROCEDURE samdbcat.persnl.spj1
    TO 'HR.MIKE', 'HR.JOE', 'HR.HILDE';
  ```

- The owner of `spj2` grants EXECUTE privilege on that SPJ to all users of the system:

  ```
  GRANT EXECUTE
    ON samdbcat.persnl.spj2
    TO PUBLIC;
  ```

GRANT SECURITY_ADMIN Statement

Considerations for GRANT SECURITY_ADMIN
Examples of GRANT SECURITY_ADMIN

The GRANT SECURITY_ADMIN statement designates a specified user as a security administrator.

```
GRANT SECURITY_ADMIN to authid
```

authid

specifies the authorization ID whom you are designating a Security Administrator. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name, enclosed in double quotes. A Guardian user number (for example, “255,255”) is not allowed. *authid* is not case-sensitive.

Considerations for GRANT SECURITY_ADMIN

Authorization Requirements

If the Security Administrator's Group is empty, only the Super ID may execute the GRANT SECURITY_ADMIN statement. Otherwise, only a Security Administrator may execute this statement.

Security Considerations

NonStop SQL/MX translates each authorization ID you specify into a 32-bit integer value and then stores the number in the system metadata tables. The stored identification number, not the characters of the authorization ID, is used to identify a Security Administrator. For this reason, care must be exercised when reusing vacated Guardian user IDs. HP recommends utilizing a dedicated Guardian user group for Security Administrators.

To prevent a Security Administrator from creating a user for themselves and granting any privilege to that user, HP strongly recommends that the function of creating users be restricted to users outside the Security Administrator's Group.

Since object owners may continue to grant privileges in the presence of Security Administrators and owner-derived grants exist distinctly from those made by Security Administrators, HP recommends that object ownership reside with an entity such as a DBA who would be expected to refrain from making owner-derived grants. HP also recommends periodic auditing of object privileges to detect and correct unauthorized grants.

Metadata Version Requirements

The GRANT SECURITY_ADMIN statement requires system metadata version 3100 or greater. If the statement is executed with lower versions of the system metadata, a SQL error 25223 is generated.

Examples of GRANT SECURITY_ADMIN

- The following example designates the Super ID as a Security Administrator:

  ```
  GRANT SECURITY_ADMIN TO "SUPER.SUPER";
  ```

- The following example designates the user, SECADMIN.USER1, as a Security Administrator:

  ```
  GRANT SECURITY_ADMIN TO "SECADMIN.USER1";
  ```

INITIALIZE SQL Statement

[Considerations for INITIALIZE SQL](#)
[Examples of INITIALIZE SQL](#)

The INITIALIZE SQL statement prepares a node to run NonStop SQL/MX.

INITIALIZE SQL is an SQL/MX extension.

```
INITIALIZE SQL
```

The INITIALIZE SQL statement creates the SQL/MX user metadata (UMD) tables and system metadata (SMD) tables in the system volume configured during installation.

If the volume is not audited, INITIALIZE SQL cannot execute. You cannot perform any database requests until SQL is initialized. If SQL is already initialized, INITIALIZE SQL returns an error.

Considerations for INITIALIZE SQL

INITIALIZE SQL is normally performed automatically by the script that installs NonStop SQL/MX. You will probably never manually perform an INITIALIZE SQL statement but it is described here for reference.

Authorization and Availability Requirements

The super ID becomes the owner of the SQL/MX user metadata (UMD) tables and system metadata (SMD) tables. The PUBLIC user has GRANT PUBLIC SELECT access on all of these tables.

Examples of INITIALIZE SQL

- This example initializes SQL on the local node:

  ```
  INITIALIZE SQL;
  ```

INSERT Statement

[Considerations for INSERT](#)
[MXCI Examples of INSERT](#)
[C Examples of INSERT](#)
[COBOL Examples of INSERT](#)

The INSERT statement is a DML statement that inserts rows in a table or view.

MXCI

```
[ ROWSET FOR INPUT SIZE rowset-size-in ]

INSERT INTO table [(target-col-list)] insert-source

target-col-list is:
   colname [,colname]...

insert-source is:
   query-expr [order-by-clause] | DEFAULT VALUES

query-expr is:
   non-join-query-expr | joined-table

non-join-query-expr is:
   non-join-query-primary | query-expr UNION [ALL] query-term

query-term is:
   non-join-query-primary | joined-table

non-join-query-primary is:
   simple-table | (non-join-query-expr)

joined-table is:
   table-ref [NATURAL] [join-type] JOIN table-ref [join-spec]
  |table-ref CROSS JOIN table-ref

table-ref is:
     table [[AS] corr [(col-expr-list)]]
   | view  [[AS] corr [(col-expr-list)]]
   | (query-expr) [AS] corr [(col-expr-list)]
   | joined-table

join-type is:
   INNER | LEFT [OUTER] | RIGHT [OUTER]

join-spec is:
   ON search-condition | rowset-search-condition
```

```
simple-table is:
     VALUES (row-value-const) [,(row-value-const)]...
   | VALUES (rowset-value-const)
   | TABLE table
   | SELECT [ALL | DISTINCT] select-list
        FROM table-ref [,table-ref]...
      | FROM ROWSET [rowset-size]
         (:array-name [,:array-name]...)
        [WHERE search-condition | rowset-search-condition]
        [SAMPLE sampling-method]
        [TRANSPOSE transpose-set [transpose-set]...
           [KEY BY key-colname]]...
        [SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
           [,colname [ASC[ENDING] | DESC[ENDING]]]...]
        [GROUP BY {colname | colnum} [,{colname |
colnum}]...]
        [HAVING search-condition | rowset-search-condition]]
        [[FOR] access-option ACCESS]
        [IN {SHARE | EXCLUSIVE} MODE]

rowset-value-const is:
     {rowset-expr |expr | NULL | DEFAULT}
   [,{rowset-expr |expr | NULL | DEFAULT}]...

access-option is:
     READ COMMITTED
   | SERIALIZABLE
   | REPEATABLE READ

order-by-clause is:
   ORDER BY {colname | colnum} [ASC[ENDING] | DESC[ENDING]]
      [,{colname | colnum} [ASC[ENDING] | DESC[ENDING]]]... ■
```

MXCI

ROWSET FOR INPUT SIZE *rowset-size-in*

rowset-size-in restricts the size of the input rowset to the specified size, which must be less than or equal to the allocated size for the rowset. *rowset-size-in* must be an integer literal (exact numeric literal, dynamic parameter, or a host variable) whose type is unsigned short, signed short, unsigned long, or signed long in C and their corresponding equivalents in COBOL. If you do not specify *rowset-size-in*, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section of the embedded SQL program. ■

table

names the user table or view in which to insert rows. *table* must be either a base table or an updatable view. To refer to a table or view, use one of these name types:

- Guardian physical name
- ANSI logical name
- DEFINE name

See Database Object Names on page 6-13.

(target-col-list)

names the columns *target-col-list* in the table or view in which to insert values. The data type of each target column must be compatible with the data type of its corresponding source value. Within the list, each target column must have the same position as its associated source value, whose position is determined by the columns in the table derived from the evaluation of the query expression (*query-expr*).

If you do not specify all of the columns in *table* in the *target-col-list*, column default values are inserted into the columns that do not appear in the list. See Column Default Settings on page 6-8.

If you do not specify *target-col-list*, row values from the source table are inserted into all columns in *table* (with the exception of a SYSKEY column). The order of the column values in the source table must be the same order as that of the columns specified in the CREATE TABLE for *table*. (This order is the same as that of the columns listed in the result table of SELECT * FROM *table*.)

insert-source

specifies the rows of values *insert-source* to be inserted into all columns of *table* or, optionally, into specified columns of *table*.

query-expr

specifies the query expression that generates the source table consisting of rows of values to be inserted into the columns named in *target-col-list*, if specified, or into all the columns of *table* by default. If there are no rows returned in *insert-source*, no rows are inserted into *table*. If *query-expr* is not a VALUES clause, the *insert-source* cannot reference either *table* or any view based on *table*, or any base table or view on which *table* is based.

The number of columns in the column list (or by default the number of columns in *table*) must be equal to the number of columns in the source table derived from the evaluation of the query expression. Further, the data type of each column in the column list (or by default each column in *table*) must be compatible with the data type of its corresponding column in the source table.

A single value within a VALUES clause can be a value expression, NULL, or DEFAULT. If you specify DEFAULT within a VALUES clause, the value inserted is the DEFAULT value defined for the target column. A value expression can also include DEFAULT as an operand; the value inserted is the expression evaluated with the DEFAULT value. For example, DEFAULT + 50 can be an expression in a row value constructor.

The use of DEFAULT in a value expression is an SQL/MX extension.

If you attempt to insert NULL into a column that is defined as NOT NULL or DEFAULT into a column that is defined with NO DEFAULT, NonStop SQL/MX returns an error.

For the description of value expressions, see Expressions on page 6-41. For the description of *query-expr*, see SELECT Statement on page 2-330.

Embed

rowset-value-const

There must be at least one rowset expression in the rowset value constructor. See the *SQL/MX Programming Manual for C and COBOL* for a discussion of semantics when rowsets of different length or rowsets and scalars are used in a rowset value constructor.

FROM ROWSET *rowset-size*

restricts the size of the rowset-derived table to the specified size, which must be less than or equal to the allocated size for the rowset. The size, if specified, immediately follows the ROWSET keyword. The size is an unsigned integer or a host variable whose value is an unsigned integer. If you do not specify the size, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section.

:*array-name* [,:*array-name*]...

specifies a set of host variable arrays. Each *array-name* can be used like a column in the rowset-derived table. Each *array-name* can be any valid host language identifier with a data type that corresponds to an SQL data type. Precede each *array-name* with a colon (:) within an SQL statement.

For more information on rowsets and host variable arrays, see the *SQL/MX Programming Manual for C and COBOL*. ■

MXCI

```
ORDER BY {colname | colnum  [ASC[ENDING] | DESC[ENDING]]
   [,{colname | colnum} [ASC[ENDING] | DESC[ENDING]]]...
```

determines the order of the rows in the source table derived from the evaluation of *query-expr* and therefore the order of insertion into *table*. The query expression is evaluated and the source table ordered before inserting any rows into the target table. Note that this option has no effect when inserting into a table with a key-sequenced physical organization.

colname

is the name *colname* of a column in a table or view that is referenced by the query expression and optionally qualified by a table, view, or correlation name; for example, CUSTOMER.CITY. If a column has been aliased to another name you must use the alias name.

colnum

specifies a column by its position *colnum* in the select list of the query expression. Use *colnum* to refer to unnamed columns, such as columns in the derived table of a query expression other than a table or view.

ASC | DESC

specifies the sort order. The default is ASC. For ordering the source table on a column that can contain null, nulls are considered equal to one another but greater than non-nulls. ■

DEFAULT VALUES

specifies a query expression of the form VALUES (DEFAULT, ...). The value of each DEFAULT is the default value defined in the column descriptor of *colname*, which is contained in the table descriptor of *table*. Each default value is inserted into its column to form a new row.

If you specify DEFAULT VALUES, you cannot specify a column list. You can use DEFAULT VALUES only when all columns in *table* have default values.

[FOR] *access-option* ACCESS

specifies the access option required for data accessed and returned in the source table derived from the evaluation of a query expression that is a SELECT statement. See Data Consistency and Access Options on page 1-8.

READ COMMITTED

specifies that any data accessed and returned in the source table derived from the evaluation of the query expression must be from committed rows.

SERIALIZABLE | REPEATABLE READ

specifies that the INSERT statement and any concurrent process (accessing the same data) execute as if the statement and the other process had run serially rather than concurrently.

The default access option is the isolation level of the containing transaction, which is determined according to the rules specified in Isolation Level on page 10-56.

IN {SHARE | EXCLUSIVE} MODE

specifies that either SHARE or EXCLUSIVE locks be used when accessing data specified by a SELECT statement or by a table reference in the FROM clause derived from the evaluation of a query expression that is a SELECT statement; and when accessing the index, if any, through which the table accesses occur.

Considerations for INSERT

Starting with SQL/MX Release 3.2, self-referencing inserts are supported. With this support, you can select the rows to be inserted from the target table in a subquery.

Statement atomicity means that a statement will either complete or be rolled back, without having to rollback a business transaction that contains multiple statements. SQL/MX will try to undo any changes to the database as a result of an insert in case a row cannot be inserted, typically because of a constraint violation such as a duplicate row.

There are some conditions where such an undo operation will cause an active transaction to be rolled back instead of just the statemement. The following are some examples where the active transaction will be rolled back:

- Parallel inserts performed by ESPs
- VSBB inserts (either explicitly enforced by the CQD INSERT_VSBB set to ON or when chosen by the optimizer)
- CQD UPD_ABORT_ON_ERROR is set to ON to force transactions to be aborted. This CQD is supported to preserve the behavior of older releases
- The underlying table has referential integrity constraints or triggers defined

For more information, see Transaction Management on page 1-13.

Use the EXPLAIN statement to check whether transactions will be rolled back or if statement atomicity will be used. For more information, see EXPLAIN Statement on page 2-208.

Authorization Requirements

INSERT requires authority to read and write to the table or view receiving the data and authority to read tables or views specified in the query expression (or any of its subqueries) in the INSERT statement.

Transaction Initiation and Termination

The INSERT statement will automatically initiate a transaction only if TRANSACTION AUTOBEGIN is set to ON. If a separate BEGIN WORK was issued, the INSERT statement operates under that transaction.

The INSERT statement will commit the transaction if TRANSACTION AUTOCOMMIT is set to ON. If AUTOCOMMIT is set to OFF, you must explicitly commit the transaction.

If a table is not audited, transactions do not apply.

Isolation Levels of Transactions and Access Options of Statements

The isolation level of an SQL/MX transaction defines the degree to which the operations on data within that transaction are affected by operations of concurrent transactions. When you specify access options for the DML statements within a transaction, you override the isolation level of the containing transaction. Each statement then executes with its individual access option.

Note. NonStop SQL/MX accepts SQL/MP keywords as synonyms for READ UNCOMMITTED, STABLE, and SERIALIZABLE.

You can explicitly set the isolation level of a transaction with the SET TRANSACTION statement. See SET TRANSACTION Statement on page 2-376. The default isolation level of a transaction is determined according to the rules specified in Isolation Level on page 10-56.

Embed It is important to note that the SET TRANSACTION statement might cause a dynamic recompilation of the DML statements within the next transaction. Dynamic recompilation occurs if NonStop SQL/MX detects a change in the transaction mode at run time compared with the transaction mode at the time of static SQL compilation. To avoid dynamic recompilation because of a change in the transaction mode, consider specifying access options for individual DML statements instead of using SET TRANSACTION. ■

Use of a VALUES Clause for the Source Query Expression

If the query expression consists of the VALUES keyword followed by rows of values, each row consists of a list of value expressions or a row subquery (a subquery that returns a single row of column values). A value in a row can also be a scalar subquery (a subquery that returns a single row consisting of a single column value).

Within a VALUES clause, the operands of a value expression can be numeric, string, datetime, or interval values; however, an operand cannot reference a column (except in the case of a scalar or row subquery returning a value or values in its result table).

Embed

Inserting From Host Variables

To insert a row from host variables, an application program moves the new values to a sequence of host variables, and then executes an INSERT statement to transfer the row of values from the host variables to the table or view.

In this situation, the query expression that defines the insert source is specified as:

VALUES (*variable-spec* [,*variable-spec*]...)

C/COBOL Each variable specification has the form:

:*variable-name* [[INDICATOR] :*indicator-name*]

The variable specification is a declared host variable with an optional indicator variable. To insert null into a database, set the indicator variable to a value less than zero. ■

For more information, see the *SQL/MX Programming Manual for C and COBOL*. ■

Requirements for Inserted Rows

Each row to be inserted must satisfy the constraints of the table or underlying base table of the view. A table constraint is satisfied if the check condition is not false—it is either true or has an unknown value.

Using Compatible Data Types

To insert a row, you must provide a value for each column in the table that has no default value. The data types of the values in each row to be inserted must be compatible with the data types of the corresponding target columns.

Inserting Character Values

Any character string data type is compatible with all other character string data types that have the same character set. For fixed length, an inserted value shorter than the column length is padded on the right with single-byte ASCII blanks (HEX 20). If the value is longer than the column length, string truncation of nonblank trailing characters returns an error, and the truncated string is not inserted.

For variable length, a shorter inserted value is not padded. As is the case for fixed length, if the value is longer than the column length, string truncation of nonblank trailing characters returns an error, and the truncated string is not inserted.

Inserting Numeric Values

Any numeric data type is compatible with all other numeric data types. If you insert a value into a numeric column that is not large enough, an overflow error occurs. If a value has more digits to the right of the decimal point than specified by the scale for the column definition, the value is truncated.

Inserting Interval Values

A value of INTERVAL data type is compatible with another value of INTERVAL data type only if the two data types are either both year-month or both day-time intervals.

Inserting Date and Time Values

DATE, TIME, and TIMESTAMP are the three SQL/MX datetime data types. A value with a datetime data type is compatible with another value with a datetime data type only if the values have the same datetime fields.

Inserting Nulls

In addition to inserting values with specific data types, you might want to insert nulls. To insert null, use the keyword NULL.

Audited and Nonaudited Tables

SQL/MX tables must be audited. You can run NonStop SQL/MX against nonaudited SQL/MP tables.

The TMF product works only on audited tables, so a transaction does not protect operations on nonaudited tables. Nonaudited tables follow a different locking and error handling model than audited tables. Certain situations such as DML error occurrences or utility operations with DML operations can lead to inconsistent data within a nonaudited table or between a nonaudited table and its indices.

To avoid problems, do not run DDL or utility operations concurrently with DML operations on nonaudited tables. When you try to delete data in a nonaudited table with an index, NonStop SQL/MX returns an error.

Considerations for self-referencing inserts

- The Halloween problem occurs when rows are updated using an index that contains a column being updated. As the column is updated, the current index entry is deleted, creating a possibility that a new index could be inserted ahead of the current entry. Similarly, when a self-referencing insert uses the primary key or an index to select rows from a table that are inserted into the same table, the newly inserted rows may also qualify the selection criteria and thus the insert statement may create an infinite loop. When this happens, the rows appear repeatedly.

  SQL/MX Release 3.2 overcomes the Halloween problem using the following methods:

  - **Blocking Plan Operator Method** - A blocking operator is introduced in the query plan to ensure that all the selected rows are read from the target table before inserting new rows. Self-referencing UPDATE, DELETE statements and UPDATE statements on primary key unique index columns use this method.
  - **DP2 Locks Method (DP2 Locks)** - DP2 ensures that newly added rows are skipped for the scan. By default, the DP2 Locks Method is used to overcome the Halloween problem.

    The optimizer chooses the Blocking Plan Operator Method over the DP2 Locks Method when the following criteria are met:

    - Estimated number of rows to be inserted (input cardinality) is greater than the lock escalation limit per partition
    - SERIALIZABLE ACCESS for the *insert-source*
    - READ UNCOMMITTED ACCESS for the *insert-source*
    - CQD BLOCK_TO_PREVENT_HALLOWEEN is ON
    - TABLELOCK is set to ON on the target table

- AUTOCOMMIT must be ON for the optimizer to choose the DP2 Locks Method, otherwise an SQL error 8107 is returned. AUTOCOMMIT is OFF for embedded SQL programs written in C, C++ or COBOL and the DP2 Locks Method cannot be used. For self-referencing INSERT statements from embedded SQL programs, SQL error 8107 is returned. To overcome this problem, ensure the optimizer chooses the Blocking Plan Operator Method by setting the CQD BLOCK_TO_PREVENT_HALLOWEEN to ON.
- Multiple SQL statements in a transaction are processed differently. If the first statement does not use self-referencing updates and locks the table, the subsequent statements cannot use self-referencing updates as they require row locks. With the default DP2 Locks Method, DP2 returns SQL error 1192 indicating a failed INSERT operation and no rows are inserted. You can overcome this problem by setting the CQD BLOCK_TO_PREVENT_HALLOWEEN to ON.

Note: The optimizer chooses either the DP2 Locks or the Blocking Plan Operator Method based on this CQD setting.

When the CQD is ON, the optimizer chooses the Blocking Plan Operator Method, and when this CQD is OFF, the optimizer chooses the DP2 Locks Method. When the CQD is set to RESET, the CQD value is reset to OFF. The default is OFF.

```
CONTROL QUERY DEFAULT BLOCK_TO_PREVENT_HALLOWEEN
{ON|OFF|RESET}
```

MXCI Examples of INSERT

- Insert a row into the CUSTOMER table and supply the value 'A2' for the CREDIT column:

  ```
  INSERT INTO sales.customer
    VALUES (4777, 'ZYROTECHNIKS', '11211 40TH ST.',
            'BURLINGTON', 'MASS.', '01803', 'A2');

  --- 1 row(s) inserted.
  ```

  Notice that the column name list is not specified for this INSERT statement. This operation works because the number of values listed in the VALUES clause is equal to the number of columns in the CUSTOMER table, and the listed values appear in the same order as the columns specified in the CREATE TABLE statement for the CUSTOMER table.

  By issuing this SELECT statement, this specific order is displayed:

  ```
  SELECT * FROM sales.customer
    WHERE custnum = 4777;

  CUSTNUM  CUSTNAME       STREET          ... POSTCODE  CREDIT
  -------  -------------  --------------      --------  ------
    4777   ZYROTECHNIKS   11211 4OTH ST. ... 01803     A2

  --- 1 row(s) selected.
  ```

- Insert a row into the CUSTOMER table:

  ```
  INSERT INTO sales.customer
    (custnum, custname, street, city, state, postcode)
    VALUES (1120, 'EXPERT MAILERS', '5769 N. 25TH PLACE',
            'PHOENIX', 'ARIZONA', '85016');

  --- 1 row(s) inserted.
  ```

  Unlike the previous example, this INSERT does not include a value for the CREDIT column, which has a default value. As a result, this INSERT must include the column name list.

  This SELECT statement shows the default value 'C1' for CREDIT:

  ```
  SELECT * FROM sales.customer
    WHERE custnum = 1120;

  CUSTNUM  CUSTNAME        STREET          ... POSTCODE  CREDIT
  -------  --------------  --------------      --------  ------
    1120   EXPERT MAILERS  5769 N. 25TH.   ... 85016     C1

  --- 1 row(s) selected.
  ```

- Insert multiple rows into the JOB table by using only one INSERT statement:

  ```
  INSERT INTO persnl.job
     VALUES (100,'MANAGER'),
  ```

```
          (200,'PRODUCTION SUPV'),
          (250,'ASSEMBLER'),
          (300,'SALESREP'),
          (400,'SYSTEM ANALYST'),
          (420,'ENGINEER'),
          (450,'PROGRAMMER'),
          (500,'ACCOUNTANT'),
          (600,'ADMINISTRATOR'),
          (900,'SECRETARY');

--- 10 row(s) inserted.
```

- The PROJECT table consists of five columns using the data types NUMERIC, VARCHAR, DATE, TIMESTAMP, and INTERVAL. Insert values by using these types:

```
INSERT INTO persnl.project
  VALUES (1000, 'SALT LAKE CITY', DATE '1996-10-02',
  TIMESTAMP '1996-12-21:08:15:00.00', INTERVAL '30' DAY);

--- 1 row(s) inserted.
```

- Suppose that CUSTLIST is a view of all columns of the CUSTOMER table except the credit rating. Insert information from the SUPPLIER table into the CUSTOMER table through the CUSTLIST view, and then update the credit rating:

```
INSERT INTO sales.custlist
  (SELECT * FROM invent.supplier
   WHERE suppnum = 10);

UPDATE sales.customer
  SET credit = 'A4'
  WHERE custnum = 10;
```

You could use this sequence in the following situation. Suppose that one of your suppliers has become a customer. If you use the same number for both the customer and supplier numbers, you can select the information from the SUPPLIER table for the new customer and insert it into the CUSTOMER table through the CUSTLIST view (as shown in the example).

This operation works because the columns of the SUPPLIER table contain values that correspond to the columns of the CUSTLIST view. Further, the credit rating column in the CUSTOMER table is specified with a default value. If you want a credit rating that is different from the default, you must update this column in the row of new customer data.

- This self-referencing INSERT statement uses the DP2 Locks Method. Look for the flag `self_referencing_update` in the explain output:

```
drop table test1;
create table test1(col1 int not null , col2 char(3),primary
key(col1) );
insert into test1 values ( 1, '100'), (2, '200');
--- 2 row(s) inserted.
prepare s1 from insert into test1 select col1 + 10, col2 from
```

```
test1 where col1 <= 3;
explain s1;

------------------------------------------------------------------
----- PLAN SUMMARY
MODULE_NAME .............. DYNAMICALLY COMPILED
STATEMENT_NAME ........... S1
PLAN_ID .................. 212204691857830064
ROWS_OUT ................ 33
EST_TOTAL_COST ........... 0.03
STATEMENT ................ insert into test1
                           select col1 + 10, col2
                           from test1
                           where col1 <= 3;


------------------------------------------------------------------
----- NODE LISTING

ROOT ======================================  SEQ_NO 6
ONLY CHILD 5
REQUESTS_IN .............. 1
ROWS_OUT ................ 33
EST_OPER_COST ............ 0
EST_TOTAL_COST ........... 0.03
DESCRIPTION
  max_card_est .......... 33
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  statement_index ........ 0
  olt_optimization ....... not used
  affinity_value  304,391,840
  upd_action_on_error .... savepoint
  xn_autoabort_interval    -1
  plan_version ....... 3,200
  self_referencing_update  dp2_locks
  MXCI_PROCESS ........... ON
  SHOWCONTROL_UNEXTERNALI  OFF
```

```
  select_list ............ %(10)
........

>>execute s1;
--- 2 row(s) inserted.
>>select * from test1;

COL1         COL2

-----------  ----

          1  100

          2  200

         11  100

         12  200

--- 4 row(s) selected.

control query default BLOCK_TO_PREVENT_HALLOWEEN 'ON';

>>control query default BLOCK_TO_PREVENT_HALLOWEEN 'ON';

--- SQL operation complete.
```

- This self-referencing INSERT statement uses the Blocking Plan Operator Method. Look for the flag `self_referencing_update` and `forced_sort` in the explain output. The forced sort causes the plan to block until all records are processed:

```
>>fc pre

>>prepare s1 from insert into test1 select col1 + 10, col2
from test1 where col1 <= 3;

..
--- SQL command prepared.

>>fc exp
>>explain s1;
..
-------------------------------------------------------------
----- PLAN SUMMARY

MODULE_NAME .............. DYNAMICALLY COMPILED
STATEMENT_NAME ........... S1
PLAN_ID .................. 212204692084904912
ROWS_OUT ................ 33
EST_TOTAL_COST ........... 0.04
STATEMENT ................ insert into test1
                           select col1 + 10, col2
                           from test1
                           where col1 <= 3;
-------------------------------------------------------------
----- NODE LISTING

ROOT ======================================  SEQ_NO 7
ONLY CHILD 6
REQUESTS_IN .............. 1
```

```
ROWS_OUT ................ 33
EST_OPER_COST ............ 0
EST_TOTAL_COST ........... 0.04
DESCRIPTION
  max_card_est .......... 33
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  statement_index ........ 0
  olt_optimization ....... not used
  affinity_value  1,687,913,920
  upd_action_on_error .... savepoint
  xn_autoabort_interval    -1
  plan_version ....... 3,200
  self_referencing_update  forced_sort
  MXCI_PROCESS ........... ON
  SHOWCONTROL_UNEXTERNALI  OFF

  BLOCK_TO_PREVENT_HALLOW  ON
  select_list ............ %(10)


>>execute s1;

--- 2 row(s) inserted.

>>select * from test1;

COL1         COL2

-----------  ----

          1  100

          2  200

         11  100

         12  200

--- 4 row(s) selected.
```

C Examples of INSERT

- Execute an INSERT statement:

```
...
EXEC SQL INSERT INTO SALES.CUSTOMER
  (CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE)
  VALUES (1120, 'EXPERT MAILERS', '5769 N.25TH PLACE',
          'PHOENIX', 'ARIZONA', '85016');
...
```

- Use host variables to insert values with an INSERT statement:

```
...
EXEC SQL INSERT INTO SALES.CUSTOMER
  (CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE)
  VALUES (:hv_custnum, :hv_custname, :hv_street,
          :hv_city, :hv_state, :hv_postcode);
...
```

- Execute an INSERT statement that includes an ORDER BY on a column that is used in an expression. The correlation name, MLT1, is the column that NonStop SQL/MX uses for the ORDER BY:

```
INSERT INTO temp1
(SELECT (UDEC1_NUNIQ - 1) as MLT1 FROM
    $SQL04.SQLDPOPS.B2PWL34)
      ORDER BY MLT1;
```

COBOL Examples of INSERT

- Execute an INSERT statement:

```
...
EXEC SQL INSERT INTO SALES.CUSTOMER
  (CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE)
  VALUES (1120, 'EXPERT MAILERS', '5769 N.25TH PLACE',
          'PHOENIX', 'ARIZONA', '85016')
END-EXEC.
...
```

- Use host variables to insert values with an INSERT statement:

```
...
EXEC SQL INSERT INTO SALES.CUSTOMER
  (CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE)
  VALUES (:hv-custnum, :hv-custname, :hv-street,
          :hv-city, :hv-state, :hv-postcode)
END-EXEC.
...
```

LOCK TABLE Statement

Considerations for LOCK TABLE
Examples of LOCK TABLE

The LOCK TABLE statement locks a table (or the underlying tables of a view) and its indexes, limiting other access to the table and its indexes while your process executes DML statements. See Database Integrity and Locking on page 1-11.

LOCK TABLE is an SQL/MX extension.

```
LOCK TABLE table IN {SHARE | EXCLUSIVE} MODE
```

table

is the name of the table or view to be locked. See Database Object Names on page 6-13.

`IN {SHARE | EXCLUSIVE} MODE`

specifies the locking mode:

SHARE	Other processes can read, but not delete, insert, or update the table or view.
EXCLUSIVE	Other processes can read with READ UNCOMMITTED access, but cannot read with READ COMMITTED or SERIALIZABLE access, and cannot delete, insert, or update the table or view.

If you request a SHARE lock on a table locked with an EXCLUSIVE lock by another user, your request waits until the EXCLUSIVE lock is released.

If you request an EXCLUSIVE lock on a table and any part of the table is locked by another user, your request waits until the lock is released, or until your lock request times out and an error message is returned.

Considerations for LOCK TABLE

Authorization Requirements

To lock a table, you must have authority to read the table. To lock a view, you must have authority to read the view but not necessarily the tables underlying the view.

Modifying Default Locking

A SELECT statement automatically acquires SHARE locks unless you specify EXCLUSIVE in the IN clause of the SELECT statement. The DELETE, INSERT, and UPDATE statements automatically acquire EXCLUSIVE locks.

You can use LOCK TABLE with the EXCLUSIVE option to force the use of EXCLUSIVE locks for a subsequent SELECT; however, keep in mind that LOCK TABLE locks the entire table.

Unlocking Locked Tables

Audited tables do not need to be explicitly unlocked. An audited table can be locked only within a transaction and is automatically unlocked when the transaction ends.

You can unlock nonaudited tables by using UNLOCK TABLE. However, locked tables are unlocked automatically when you issue COMMIT WORK or ROLLBACK WORK to end a user-defined transaction or when your MXCI session ends. Only SQL/MP tables can be nonaudited.

Effect of AUTOCOMMIT Option

At the start of an MXCI session, the AUTOCOMMIT option is ON by default. When this option is ON, NonStop SQL/MX automatically commits any changes, or rolls back any changes, made to the database at the end of statement execution. When you issue a LOCK TABLE statement in MXCI without turning off AUTOCOMMIT, NonStop SQL/MX locks the table temporarily, and then commits the transaction at the end of the LOCK TABLE statement and releases the locks. If you use LOCK TABLE in MXCI, turn off AUTOCOMMIT by using the SET TRANSACTION statement. See SET TRANSACTION Statement on page 2-376.

Partitions and Indexes

LOCK TABLE attempts to lock all partitions and indexes of any table it locks. If a partition or index is not available or if the lock request times out, LOCK TABLE displays a warning and continues to request locks on other partitions and indexes.

Examples of LOCK TABLE

- Lock an audited table with an EXCLUSIVE lock (at a time when few users need access to the database) to perform a series of updates:

```
BEGIN WORK;

LOCK TABLE persnl.employee
  IN EXCLUSIVE MODE;

UPDATE persnl.employee
  SET salary = salary * 1.05
  WHERE jobcode <> 100;

COMMIT WORK;
```

  COMMIT WORK automatically unlocks the table when it ends the transaction.

- Delete all rows of the JOB table that have a job code that is not assigned to any employee:

```
BEGIN WORK;
--- SQL operation complete.

LOCK TABLE persnl.job
  IN EXCLUSIVE MODE;
--- SQL operation complete.

LOCK TABLE persnl.employee
  IN SHARE MODE;
--- SQL operation complete.

DELETE FROM persnl.job
WHERE jobcode NOT IN
  (SELECT DISTINCT jobcode
   FROM persnl.employee);
--- 1 row(s) deleted.

COMMIT WORK;
--- SQL operation complete.

UNLOCK TABLE persnl.job;
--- SQL operation complete.
```

In this example, suppose that the JOB table is nonaudited and you want locks held for several transactions. Because the EMPLOYEE table is audited and you are locking it, you define a transaction. At the end of the transaction, the EMPLOYEE table lock is released by the system. You must use the UNLOCK TABLE command to release the lock on the JOB table because the table is nonaudited.

MODIFY Utility

Considerations for MODIFY
Examples of MODIFY

MODIFY is a syntax-based utility that can be executed through MXCI that enables database administrators to perform partition operations on range and hash partitions of SQL/MX tables and indexes. Depending on the type of operation you are performing, MODIFY can be run as an online or offline operation. See Considerations for MODIFY on page 2-290 for details about limitations on online operations.

The forms of the MODIFY statement are:

- Reuse an Existing Partition of a Range Partitioned Table
- Manage Partitions of Range Partitioned Tables and Indexes
- Manage Partitions of Hash Partitioned Tables and Indexes
- Manage System-Clustered Tables
- Managing a Sequence Generator
- Renaming Guardian Location of Partitions of Tables, Indexes or Sequence Generators

Reuse an Existing Partition of a Range Partitioned Table

Use MODIFY to reuse an existing range partition of a table by setting the FIRST KEY values of the partition to new values. You can optionally remove existing data in the partition to be reused. No data can exist in the new key range. Only offline operations are supported.

The REUSE form of MODIFY is:

```
MODIFY TABLE [[catalog.]schema.]table
   REUSE [PARTITION] WHERE partition-identification
   WITH [KEY=] key-value
   [[NO] PURGEDATA]

partition-identification is:
    LOCATION [\node.]$volume[.subvolume.file-name]
   | NAME partition-name
   | [KEY=] {FIRST | LAST} PARTITION
   | [KEY=] key-value

key-value is:
VALUE (column-value [,column-value]...)
```

[[catalog.]schema.]table

specifies the name of the table. If you do not specify the schema and catalog name parts, MODIFY uses the current default catalog and schema of your MXCI session.

partition-identification

describes the partition.

LOCATION [*node.*]*$volume*[.*subvolume*.*file-name*]
| NAME *partition-name*
| [KEY=] {FIRST | LAST} PARTITION
| [KEY=] *key-value*

is a location for a partition or a name for a partition or the partitioning key (the FIRST KEY) value (*key-value*) of a partition, to be modified.

If the partition is the primary partition, you can also specify the partition using the FIRST PARTITION phrase. If the partition is the last partition in the list of partitions of the table or index, you can use the LAST PARTITION phrase. You can use either phrase to specify a partition if it is the only partition of the object.

[*node*.]$*volume*
| [*node*.]$*volume*.*subvolume*.*file-name*

is the physical location of a partition. If you do not specify the file name, a volume can be used only once for a given table or index.

node can be either the local node or a remote node. If you do not specify *node*, the default is the Guardian system named in your =_DEFAULTS define.

partition-name

is a SQL identifier for a partition.

key-value

is the key value of a partition to be modified.

VALUE (*column-value* [,*column-value*]...)

is the boundary values for the partition to be modified. You can identify a partition by its partitioning key value *key-value*. You can omit the values of the suffix columns in the FIRST KEY value provided that the specified column values can adequately identify the partition. If you omit a *column-value*, MODIFY uses either the low value or the high value of the corresponding partitioning key column, depending on whether the column stores data in ascending or descending order.

WITH [KEY=] *key-value*

assigns the new partitioning key value *key-value* to the specified partition. If you omit the values of the suffix columns, MODIFY uses the default value, either the low or high value of the corresponding partitioning key column, depending on whether the column contains data in ascending or descending order. You can omit the column values only on the right of the list.

`[[NO] PURGEDATA]`

specifies whether the existing data in the specified partition is removed. If the partition contains data and you do not explicitly specify the PURGEDATA option, MODIFY returns an error. The default is NO PURGEDATA.

Manage Partitions of Range Partitioned Tables and Indexes

Use MODIFY to manage range partitions of SQL/MX tables and indexes. You must manage tables and indexes separately regardless of their relationship. Both offline and online operations are supported. See Considerations for MODIFY on page 2-290 for details about limitations on online operations.

The form of MODIFY for range partitioned tables and indexes is:

```
MODIFY { TABLE | INDEX } [[catalog.]schema.] object
   { drop | add | move }

drop is:
  DROP [PARTITION] WHERE partition-identification

add is:
  ADD [PARTITION] WHERE add-move-boundary-range
  [TO] LOCATION new-partition
  [NAME new-partition-name]
  [partition-size]
  [RECLAIM | NO RECLAIM]
  [with-shared-access]

move is one of:
  MOVE [PARTITION] [WHERE partition-identification]
  [TO] LOCATION new-partition
  [NAME new-partition-name]
  [partition-size]
  [RECLAIM | NO RECLAIM]
  [with-shared-access]

  MOVE [PARTITION]
   WHERE {add-move-boundary-range | partition-identification}
  [TO] {PREVIOUS | NEXT} PARTITION
  [RECLAIM | NO RECLAIM]
  [with-shared-access]

partition-identification is:
   LOCATION [\node.]$volume[.subvolume.file-name]
 | NAME partition-name
 | [KEY=] {FIRST | LAST} PARTITION
 | [KEY=] key-value

add-move-boundary-range is:
    [KEY=] FIRST KEY UPTO [KEY=] key-value
  | [KEY=] key-value [THRU [KEY=] LAST KEY]

key-value is:
   VALUE (column-value [,column-value]...)

partition-size is:
   partition-extent-size [ MAXEXTENTS num-extents ]

partition-extent-size is:
   EXTENT {  pri-ext-size                                   }
          { (pri-ext-size, sec-ext-size) }

with-shared-access is:
   WITH SHARED ACCESS [commit-options]
```

```
commit-options is:
   { COMMIT [WORK] [{ WHEN READY } | { AFTER time } | {
BEFORE time }] [on-error] }

on-error is:
  { ONCOMMITERROR
   { COMMIT [WORK] [{ WHEN READY } | { AFTER time } | {
BEFORE time }]}
  | { ROLLBACK [WORK] }
  }
```

`[[`*`catalog`*`.]`*`schema`*`.]`*`object`*

is the name of the range partitioned object. If you do not specify the schema and catalog name, NonStop SQL/MX uses the default catalog and schema of your MXCI session. *`object`* is a table or an index, depending on the TABLE or INDEX keyword.

`DROP [PARTITION] WHERE `*`partition-identification`*

is a request to drop a range partition. The specified partition must be empty.

`partition-identification`

describes the partition.

```
LOCATION [\node.]$volume[.subvolume.file-name]
| [KEY=] {FIRST | LAST} PARTITION
| [KEY=] key-value
```

is a location for a partition or the partitioning key (the FIRST KEY) value *`key-value`* of a partition to be moved or dropped.

When you drop a partition, its key range is merged into the previous partition unless the first partition is dropped. If you drop the first partition, its key range is merged into the next partition.

```
    [\node.]$volume
|   [\node.]$volume.subvolume.file-name
```

is the physical location of a partition. If you do not specify the file name, only one partition can exist for the given data source.

node can be either the local node or a remote node. If you do not specify *node*, the default is the Guardian system named in your =_DEFAULTS define.

```
ADD [PARTITION] WHERE add-move-boundary-range
    [TO] LOCATION new-partition
    [NAME new-partition-name]
    [partition-size]
```

specifies a request to split a range of data in an existing partition (either the beginning part or the last part) and then move it to a new partition. Data can exist in the range being added.

An operation is a prefix split if the range of data begins from the top of the existing partition. An operation is a postfix split if the range ends at the bottom of the partition.

add-move-boundary-range

is the boundary range.

```
  [KEY=] FIRST KEY UPTO [KEY=] key-value
| [KEY=] key-value [THRU [KEY=] LAST KEY]
```

specifies the partitioning range *add-move-boundary-range* of a partition to be split and then added to a new partition.

You can specify the partitioning range of a new partition to be added by splitting off the beginning or end of an existing partition with the FIRST KEY (start key value) up to, but not including, a key value *key-value* in the current partition, or by a key value *key-value* in the current partition through the LAST KEY (end key value). THRU [KEY=] LAST KEY is optional.

When you specify *key-value*, you can omit the values of the suffix columns provided that the specified column values can adequately identify the partition. If you omit *column-value*, MODIFY uses the default value, either the low or high value of the corresponding partitioning key column, depending on whether the column contains data in ascending or descending order.

[TO] LOCATION *new-partition*

specifies the location of the new partition.

new-partition

specifies a disk volume or a Guardian file for the new partition. If you use disk volume syntax, MODIFY generates the file suffix name part. The specified new partition can be on the local system or a remote system.

new-partition-name

is a SQL identifier for a partition.

```
MOVE [PARTITION] [WHERE partition-identification]
   [TO] LOCATION new-partition
   [NAME new-partition-name]
   [partition-size]
```

is a request to move an existing partition to a new location. *partition-identification* is optional only if the table or index has only one partition.

```
MOVE [PARTITION] WHERE
   {partition-identification | add-move-boundary-range}
   [TO] {PREVIOUS | NEXT} PARTITION
```

is a request to merge part or all of an existing partition to an adjacent existing partition. You can specify an entire partition using the *partition-identification* clause.

partition-identification

describes the partition.

```
   LOCATION [\node.]$volume[.subvolume.file-name]
|  NAME partition-name
|  [KEY=] {FIRST | LAST} PARTITION
|  [KEY=] key-value
```

is a location for a partition, or the partitioning key (the FIRST KEY) value (*key-value*) of a partition, to be modified.

If the partition is the primary partition, you can also specify the partition using the FIRST PARTITION phrase. If the partition is the rightmost partition in the list of partitions of the table, you can use the LAST PARTITION phrase. You can use either phrase to specify a partition if it is the only partition of the object.

```
   [\node.]$volume
|  [\node.]$volume.subvolume.file-name
```

is the physical location of a partition. If you do not specify the file name, only one partition can exist for the given data source.

node can be either the local node or a remote node. If you do not specify *node*, the default is the Guardian system named in your =_DEFAULTS define.

You can use the *add-move-boundary-range* clause to specify a range of data in a partition (either the beginning part or the last part) to be split and then merged into an adjacent and existing partition.

An operation is a prefix merge if the range of data begins from the top of the existing partition. For a prefix merge operation, you can specify only the TO PREVIOUS PARTITION clause. The split partition cannot be the primary partition.

An operation is a postfix merge if the range of data ends at the bottom of the partition. You can specify only the TO NEXT PARTITION clause. The split partition cannot be the last partition (the rightmost partition in the list).

partition-size

is the size of the new partition.

partition-extent-size [MAXEXTENTS *max-extents*]

is the size of the new partition. You can specify the sizes of the primary and secondary extents and the maximum number of extents. If you do not specify MAXEXTENTS, MODIFY uses the value of the source partition.

partition-extent-size

is the extent size of the new partition.

```
EXTENT {  ext-size                                 }
          { (pri-ext-size, sec-ext-size) }
```

ext-size is an unsigned integer value. You can specify it as the size for both primary and secondary extents of the new partition. You can specify the size of the primary extent and secondary extents separately. If you do not specify EXTENT, MODIFY uses the extent size values of the source partition.

See EXTENT on page 8-6 and MAXEXTENTS on page 8-7.

WITH SHARED ACCESS [*commit-options*]

specifies that the operation is an online operation. If you do not specify *commit-options*, the default is COMMIT WHEN READY ONCOMMITERROR ROLLBACK WORK.

```
COMMIT [ WORK ] [ WHEN READY          ]  [on-error]
                [ {  AFTER time   } ]
               [ {  BEFORE time } ]
```

time

is the time at which the commit phase should occur. The *on-error* clause specifies what will happen if the Commit Phase fails with a retryable error. If the time has already passed, MODIFY returns an error.

time is a quoted string datetime literal.

WHEN READY	Commit phase should occur at the earliest possible time.
AFTER *time*	Commit phase should occur after *time*.
BEFORE *time*	Commit phase should occur before *time*.

`ROLLBACK [WORK]`

specifies that the operation should be terminated. The effect is the same as issuing a separate RECOVER command with the CANCEL option. ROLLBACK WORK might only be specified in the last <on-error> clause.

`ONCOMMITERROR` *`commit-options`*

specifies what action SQL/MX should take if a retryable error occurs during Commit Phase. Retryable errors include file in use, lock request timeouts, resource unavailability, and BEFORE or AFTER time window misses.

A nonretryable error always causes SQL/MX to cancel changes to the database and terminate the operation, no matter what you specify in the ONCOMMITERROR option.

ONCOMMITERROR is recursive because it appears within a COMMIT option and specifies another COMMIT option. You can specify up to three COMMIT options on a single statement; specifying four or more causes an error.

`RECLAIM | NO RECLAIM`

specifies whether SQL/MX should automatically start ORSERV processes to reclaim unused freespace in affected partitions (RECLAIM) or whether the user must manually perform FUP RELOAD operations (NO RECLAIM). Partitions that contain unused freespace have the UNRECLAIMEDSPACE (F) flag set in the file label. Until the freespace is reclaimed, the flag remains set and any new MODIFY, DUP, or BACKUP operation you attempt to perform on the object will fail with error 20290 (operation still in progress). DML operations can be performed on the object, but all other operations will fail. If omitted, the default for range partitioned objects is RECLAIM. The option will be ignored in situations where MODIFY does not need to reclaim freespace.

Manage Partitions of Hash Partitioned Tables and Indexes

Use MODIFY to manage hash partitions of SQL/MX tables and indexes. You can drop only the last partition. You must manage tables and indexes separately regardless of whether they are related.

The form of MODIFY for hash partitioned tables and indexes is:

```
MODIFY {TABLE | INDEX} [[catalog.]schema.]object
  {drop | move | add}

drop is:
DROP [PARTITION] WHERE partition-identification
[RECLAIM | NO RECLAIM]
  [with-shared-access]

move is:
MOVE [PARTITION] [WHERE partition-identification]
[TO] LOCATION new-partition
[NAME new-partition-name]
[partition-size]
[RECLAIM | NO RECLAIM]
  [with-shared-access]

add is:
ADD [PARTITION] [TO] LOCATION new-partition
[NAME new-partition-name]
[partition-size]
[RECLAIM | NO RECLAIM]
  [with-shared-access]

partition-identification is:

   LOCATION [\node.]$volume[.subvolume.file-name]
 | NAME partition-name
 | {FIRST | LAST} PARTITION
 | [KEY=] VALUE(partition-number)
```

```
new-partition is:

  LOCATION [\node.]$volume[.subvolume.file-name]

<with-shared-access> is:
   WITH SHARED ACCESS [commit-options]

commit-options is:
   { COMMIT [WORK] [{ WHEN READY } | { AFTER time } | {
BEFORE time }] [on-error] }

on-error is:
  { ONCOMMITERROR
   { COMMIT [WORK] [{ WHEN READY } | { AFTER time } | {
BEFORE time }]}
  | { ROLLBACK [WORK] }
  }
```

[[*catalog*.]*schema*.]*object*

is the name of the object. If you do not specify the schema and catalog name, MODIFY uses the current default schema and catalog of your MXCI session. *object* is a table or an index, depending on the TABLE or INDEX keyword.

DROP [PARTITION] WHERE *partition-identification*

drops a hash partition. The specified partition must be the last partition (the rightmost partition in the partition array).

partition-identification

describes the partition.

```
LOCATION [\node.]$volume[.subvolume.file-name]
|  NAME partition-name
|  [KEY=] {FIRST | LAST} PARTITION
|  [KEY=] VALUE(partition-number)
```

is the location for a partition, or the partitioning key (the FIRST KEY) value (*key-value*) of a partition, to be dropped.

If you use the LOCATION clause, you must identify the last partition of the table or index.

If you use *partition-number*, it must an unsigned integer and range from 0 to n-1, where n is the number of partitions. VALUE(0) represents the first partition, VALUE(1) represents the partition adjacent to the first partition, and so on. VALUE(n-1) represents the last partition.

When you drop a hash partition, data from that partition is redistributed to the remaining partitions.

[*node*.]$*volume*
| [*node*.]$*volume*.*subvolume*.*file-name*

is the physical location of a partition. If you do not specify the file name, only one partition can exist for the given data source.

node can be either the local node or a remote node. If you do not specify *node*, the default is the Guardian system named in your =_DEFAULTS define.

```
MOVE [PARTITION] [WHERE partition-identification]
 [TO] LOCATION new-partition
 [NAME new-partition-name]
 [partition-size]
```

Moves an existing hash partition to a new location. You can define the size of the new partition using the optional *partition-size* clause. Otherwise, the values of the primary partition apply. *partition-identification* is optional only when the object has only one partition.

partition-identification

describes the partition.

```
 LOCATION [\node.]$volume[.subvolume.file-name]
|  NAME partition-name
|  [KEY=] {FIRST | LAST} PARTITION
|  [KEY=] VALUE(partition-number)
```

is the location for a partition, or the partitioning key (the FIRST KEY) value (*key-value*) of a partition, to be moved.

If you use *partition-number*, it must range from 0 to n-1, where n is the number of partitions. VALUE(0) represents the first partition, VALUE(1) represents the partition adjacent to the first partition, and so on. VALUE(n-1) represents the last partition.

```
   [\node.]$volume
|  [\node.]$volume.file-name
```

is the physical location of a partition. If you do not specify the file name, only one partition can exist for the given data source.

node can be either the local node or a remote node. If you do not specify *node*, the default is the Guardian system named in your =_DEFAULTS define.

[TO] LOCATION *new-partition*

specifies the location of the new partition.

new-partition

is a disk volume or a Guardian file for the new partition. If you use disk volume syntax, MODIFY generates the file suffix name part. The specified new partition can be on the local system or a remote system.

new-partition-name

is an SQL identifier for the new partition.

partition-size

is the size of the new partition.

partition-extent-size [MAXEXTENTS *num-extents*]

is the size of the new partition. You can specify the sizes of the primary and secondary extents and the maximum number of extents. If you do not specify MAXEXTENTS, MODIFY checks all partitions and uses the value of the source partition.

partition-extent-size

is the extent size of the new partition.

```
EXTENT {  ext-size                        }
       { (pri-ext-size, sec-ext-size) }
```

ext-size is an unsigned integer value. You can specify it as the size for both primary and secondary extents of the new partition. You can specify the size of the primary extent and secondary extents separately. If you do not specify EXTENT, MODIFY uses the extent size value of the source partition.

See EXTENT on page 9-6 and MAXEXTENTS on page 9-7.

```
ADD [PARTITION]
   [TO] LOCATION new-partition [partition-size]
```

adds a hash partition to the table or index.

[TO] LOCATION *new-partition*

specifies the location of the new partition.

new-partition

is a disk volume or a Guardian file for the new partition. If you use disk volume syntax, MODIFY generates the file suffix name part. The specified new partition can be on the local system or a remote system.

The new partition becomes the last partition of the table or index.

new-partition-name

is an SQL identifier for the new partition.

partition-size

is the size of the new partition.

partition-extent-size [MAXEXTENTS *num-extents*]

is the size of the new partition. You can specify the sizes of the primary and secondary extents and the maximum number of extents. If you do not specify MAXEXTENTS, MODIFY uses the largest maxextents size possible using the combination of primary, secondary and max extent values.

partition-extent-size

is the extent size of the new partition

```
EXTENT {  ext-size                                }
      { (pri-ext-size, sec-ext-size) }
```

ext-size is an unsigned integer value. You can specify it as the size for both primary and secondary extents of the new partition. You can specify the size of the primary extent and secondary extents separately. If you do not specify EXTENT, MODIFY uses the largest extent size possible using the combination of primary, secondary and max extent values.

See EXTENT on page 9-6 and MAXEXTENTS on page 9-7.

When you add a hash partition, a subset of data from existing partitions is redistributed to the new partition.

WITH SHARED ACCESS [*commit-options*]

specifies that the operation is an online operation. If *commit-options* is omitted, the effect is the same as specifying COMMIT WHEN READY TIMEOUT DEFAULT ONCOMMITERROR ROLLBACK WORK.

```
COMMIT [WORK] [ WHEN READY         ]  [on-error]
         [{ AFTER time  }]
         [{ BEFORE time }]
```

specifies the time at which the Commit Phase should occur. COMMIT WHEN READY specifies that the Commit Phase should occur at the earliest possible time. COMMIT AFTER *time* specifies that the Commit Phase should occur after the given *time*. COMMIT BEFORE <time> specifies that the Commit Phase should occur before the given <time>. The *on-error* clause specifies what should happen if the Commit Phase fails with a retryable error. If omitted, the effect is the same as specifying TIMEOUT DEFAULT ONCOMMITERROR

ROLLBACK WORK. *time* is a Datetime value. Example of *time*: '2005-02-16 14:00:00'

`ROLLBACK [WORK]`

specifies that the operation should be terminated. The effect is the same as issuing a separate RECOVER command with the CANCEL option. ROLLBACK WORK may only be specified in the last *on-error* clause.

`ONCOMMITERROR` *commit-options*

specifies what action SQL/MX should take if a retryable error occurs during Commit Phase. Retryable errors include file in use, lock request timeouts, resource unavailability, and BEFORE/AFTER time window misses.

A nonretryable error always causes SQL/MX to cancel changes to the database and terminate the operation, no matter what you specify in the ONCOMMITERROR option.

ONCOMMITERROR is recursive because it appears within a COMMIT option and specifies another COMMIT option. You can specify up to three COMMIT options on a single statement; specifying four or more causes an error.

`RECLAIM | NO RECLAIM`

specifies whether SQL/MX should automatically start ORSERV processes to reclaim unused freespace in affected partitions (RECLAIM) or whether the user must manually perform FUP RELOAD operations (NO RECLAIM). Partitions which contain unused freespace have the UNRECLAIMEDSPACE (F) flag set in the file label. Until the freespace is reclaimed the flag remains set and any new MODIFY, DUP, or BACKUP operation you attempt to perform on the object will fail with error 20290 (operation still in progress). DML operations can be performed on the object, but all other operations will fail. If omitted, the default for hash partitioned objects is NO RECLAIM. The option will be ignored in situations where MODIFY does not need to reclaim freespace.

Manage System-Clustered Tables

A system-clustered table has no primary key and no STORE BY clause. Its primary key defaults to the SYSKEY.

You can use MODIFY to move the existing partition of a system-clustered object to a new location. Note that a system-clustered table can have only a single partition. Only offline partition operations are supported.

The form of MODIFY for system-clustered objects is:

```
MODIFY TABLE [[catalog.]schema.]table

MOVE [PARTITION]
  [TO]

{ new-location [new-name] [partition-size] | new-name}

new-location is:
LOCATION [\node.]$volume[.subvolume.file-name]

new-name is:
NAME partition-name
```

`[[`*`catalog`*`.]`*`schema`*`.]`*`table`*

is the name of the system-clustered table. If you do not specify the schema and catalog name, NonStop SQL/MX uses the current default catalog and schema of your MXCI session.

`[\`*`node.`*`]`*`$volume`*`[.`*`subvolume`*`.`*`file-name`*`]`

is a disk volume or a Guardian file for the new partition. If you use disk volume syntax, MODIFY generates the file name. *node* can be either the local node or a remote node. If you do not specify *node*, the default is the Guardian system named in your =_DEFAULTS define.

`partition-size`

is the size of the new partition.

`partition-extent-size`` [ MAXEXTENTS `*`num-extents`*` ]`

is the size of the new partition. You can specify the sizes of the primary extent and secondary extents of the partition, and you can specify the maximum number of extents the partition can have. If you do not specify MAXEXTENTS, MODIFY uses the value of the primary partition.

`partition-extent-size`

is the extent size of the new partition

```
EXTENT {  ext-size                                }
       { (pri-ext-size, sec-ext-size) }
```

`ext-size` is an unsigned integer value. You can specify it as the size for both primary and secondary extents of the new partition. You can specify the size of the

primary extent and that of secondary extents separately. If you do not specify EXTENT, MODIFY uses the extent size values of the largest partition.

See EXTENT on page 9-6 and MAXEXTENTS on page 9-7.

Managing a Sequence Generator

You can use MODIFY to move the existing partition of a sequence generator to a new location. A sequence generator can have only a single partition. Only offline partition operations are supported.

The MODIFY form for sequence generator is:

```
MODIFY SEQUENCE [[catalog.]schema.]sequence
MOVE [ PARTITION ] [ TO ] new-location

new-location is:
 LOCATION [\node.]$volume[.sub-volume.file-name]
```

`[[catalog.]schema.]sequence`

specifies the ANSI name of the sequence generator. If you do not specify the schema and the catalog name, SQL/MX uses the current catalog and schema.

`[\node.]$volume[.sub-volume.file-name]`

specifies the disk volume or a Guardian file for the new location of the sequence generator. If you use the disk volume syntax, MODIFY generates a file name. The *node* can either be the local or a remote node. If you do not specify a *node*, the default is the Guardian system named in your =_DEFAULTS define.

Renaming Guardian Location of Partitions of Tables, Indexes or Sequence Generators

You can use MODIFY to rename the Guardian filename part at one or more locations of an existing table, index or a sequence generator. However, you cannot rename the system, volume, or subvolume part of the Guardian location.

The MODIFY form to rename the Guardian location of partitions of a table, or index:

```
MODIFY { TABLE | INDEX | SEQUENCE } [[catalog.]schema.]object
RENAME [ WHERE ] LOCATION
  { rename-spec | ( rename-spec [, rename-spec ] ... ) }
  [OUTFILE oss-file [CLEAR]]

rename-spec is:
  { simple-rename-spec
  | pattern-map-spec }

simple-rename-spec is: location TO file-name-part

pattern-map-spec is: pattern MAP NAMES TO target-map  }

location is:
  The Guardian file name of a source location to rename.

file-name-part is:
  The Guardian file name part to rename the source location
to.

pattern is:
  Standard Guardian file name pattern.

target-map is:
  an 8-character specification of how matching source file
names will be renamed. It consists of letters, digits, and
(?) characters.
```

`[[`*`catalog`*`.]`*`schema`*`.]`*`object`*

is the ANSI name of the object. The object can either be a table, index or sequence generator depending on the keyword TABLE, INDEX or SEQUENCE. If you do not specify the schema and the catalog name, the default catalog and schema names are used.

`rename-spec`

specifies how source files will be renamed to the corresponding target files. The operation uses *`rename-spec`* to rename one or more source files to the corresponding target file names.

location

specifies the Guardian file name of an individual location to be renamed. If not fully qualified Guardian defaults apply.

file-name-part

is the Guardian file name part of the renamed location.

Pattern

is a standard Guardian file name pattern which is specified to select those partitions from the table that need to be renamed. If the Guardian file name pattern is not fully qualified, the Guardian defaults apply.

target-map

is used to construct Guardian file name parts for the target location names. Each target map position corresponds to the same position in the source file name. Letters and digits will appear unchanged in the target file name. However, question mark characters (?) will be substituted with the characters at the same position in the source file name. A mapping specification has eight characters, each of which can be:

- A digit in the range 0 through 9 or a letter in the range A through Z. The target file name will contain these digits or letters in the same position as mentioned in the target-map.

A question mark (?), the target file name will contain the characters from the source file name at the corresponding positions as (?) in the target-map.

For more details, see examples of MODIFY.

Note. An error in target mapping can produce file names that already exist or a pattern that does not select any partitions.

oss-file

is the OSS path name of a log file that records the outcome of the operation. This is an optional file.

Note. If the log file already exists, the output is appended to the existing content. If the CLEAR option is used, the output overwrites the existing content.

Note. RENAME changes the redefinition timestamp of the affected table or index.

Considerations for MODIFY

- You can run MODIFY as either an offline or online operation. You can perform online operations while the partition is being used by another application. You can perform offline operations only on partitions that are not being used by other applications or that are being used with READ access. WRITE access is prohibited.

- Most of MODIFY partition management operations ignore triggers. However, the REUSE form of MODIFY returns an error if you use the PURGEDATA option on a table with DELETE triggers and if there is data in the partition:

```
*** ERROR[20294] The partition cannot be reused because the
partition contains data and the table has an enabled DELETE
trigger.
```

- If the MODIFY operation fails, use the RECOVER utility to undo or resume the failed partition operation. For details, see Checking DDL Locks on page 2-9.

- MODIFY records the status of the operation in the DDL_LOCKS metadata table. You can query this table to determine the progress of MODIFY operation.

Status	Description
1	The DDL Lock row is created.
2	The target partition is created.
3	Data is copied.
4	Copying of data is in progress.
5	Partition is marked as offline.
6	Partition is purgedata'ed and marked corrupt.
7	ORSERV is about to be started for MODIFY.
8	ORSERV is started for MODIFY.
9	ORSERV is about to be started for RECOVER
10	ORSERV is started for RECOVER.

Online Partition Management

MODIFY supports online partition management for range partitioned tables and indexes where the partitioning key is a prefix of the clustering key. Other processes can read and write the object while it is being repartitioned, except during a short period at the end when file labels and metadata are updated.

Online partition management is not supported for the following object types:

- System-clustered tables
- Range partitioned tables and indexes where the partitioning key is not a prefix of the clustering key
- Sequence generators

Offline Partition Management for Range Partitions

MODIFY supports these offline partition management operations for range partitions:

- Adding a new empty partition.
- Dropping an existing empty partition.
- Moving an existing partition to a new location.
- Splitting an existing partition and then moving the first or last part of the data to a new partition.
- Splitting an existing partition and then merging the first or last part of the data to an existing adjacent partition.
- Merging two adjacent partitions into one.
- Reusing an existing partition by setting the FIRST KEY values of the partition to new values. You can optionally remove existing data in the partition to be reused.

Offline Partition Management for Hash Partitions

MODIFY supports these offline partition management operations for hash partitions:

- Adding a new hash partition and rebalancing data (that is, redistributing existing data to all partitions, including the new partition)
- Dropping an existing hash partition and rebalancing data
- Moving an existing hash partition to a new location

Offline Partition Management for System-Clustered Partitions

MODIFY supports moving an entire system-clustered partition to a new location.

Offline Partition Management for Sequence Generators

MODIFY supports moving a sequence generator to a new location.

Renaming Guardian Locations of Partitions of Tables, Indexes or Sequence Generators

You can use the rename option to rename Guardian locations of partitions of a table, index or a sequence generator. The following are the prerequisites for a successful rename:

- The table, index, or sequence generator and all the partitions must be available.
- All specified locations must belong to the table, index or sequence generator. If the pattern form of `rename-spec` is used, the pattern must match at least one of the partitions of the table, index or sequence generator.
- The target Guardian locations must not exist.
- For all partitions of the target object, file labels must be available.

- For a Guardian rename of an index, file labels for all partitions of base table of the index must be available.

MODIFY and Indexes

If there are no indexes on a table, the reuse form of MODIFY purges data from the partition. If there are existing indexes, MODIFY performs a DELETE operation to remove the index data, which can take some time to complete.

MODIFY and TMF

Many partition management requests require movement of massive amounts of data. Because these operations might take longer than the set TMF time limit whose default is two hours, operations involving data movement are performed in multiple transactions.

Specifying the Number of Rows per Transaction

To specify the number of rows to be copied in a transaction, use the CONTROL QUERY DEFAULT statement or insert an entry to the SYSTEM_DEFAULTS table. For offline partition operations, use the PM_OFFLINE_TRANSACTION_GRANULARITY attribute. For online partition operations, use the PM_ONLINE_TRANSACTION_GRANULARITY attribute.

The setting in the SYSTEM_DEFAULTS table applies to all partition operations in the current node unless you override it by using a CONTROL QUERY DEFAULT statement. You can issue the statement from MXCI, and the setting from this statement applies only to subsequent requests within the same MXCI.

If the attribute neither appears in the SYSTEM_DEFAULTS table nor is specified using a CONTROL QUERY DEFAULT statement, MODIFY uses the value 5000 for offline partition operations and 400 for online partition operations.

Default Value for Offline Partition Operations

For offline partition operations, MODIFY locks entire source and target partitions, so DP2 lock escalation is not an issue. In general, MODIFY runs more efficiently when you specify a larger value. You should choose this value with care. If it is too large, the transaction might abort because of the two-hour TMF time limit or the size of the audit trails. You also need to take the row size into consideration when choosing the number of rows because the product of the row size and *num-of-rows* gives the amount of data to be copied in each transaction.

You can temporarily increase the TMF time limit and the size of the audit trail to allow the operations to complete with a larger *num-of-rows*. However, increasing TMF limits degrades system performance and increases disk space usage for the audit trail.

Default Value for Online Partition Operations

For online partition operations, avoid choosing a value greater than 500, because DP2 escalates locking from selected rows to the entire partition if the partition has more than 511 row and file locks.

Concurrency and Timeout Considerations

When you use MODIFY, avoid long-running concurrent transactions on the same object. Concurrency issues arise in two phases: during the data movement phase and during the commit phase.

During the data movement phase, if MODIFY is writing to an existing partition, MODIFY obtains row locks on data as it is written. If a concurrent application is also writing to the same partition, contention can occur. Either MODIFY or the application might experience timeouts if they each seek to access a row the other has locked. This situation is especially true if the application holds so many locks that DP2 attempts to escalate to a file lock or if the application transaction is long-running. If MODIFY times out, the command is terminated.

During the commit phase, MODIFY attempts to obtain exclusive locks on all partitions to update file labels. Again, if concurrent applications hold locks for long durations, MODIFY times out in its attempt, and the MODIFY command fails.

Rename

The Guardian rename operation affects the file labels and the metadata. TMF recovery is limited after the rename operation is performed. Rename of a location invalidates TMF online dumps of that location (data fork and RFork). File recovery cannot cross a rename operation.

TMF backout and volume recovery are both fully supported after the rename operations are performed.

Note. To retain recoverability, HP recommends making new TMF online dumps, immediately after a Guardian rename operation is performed.

MODIFY and RDF

Rename

RDF does not replicate the file label changes occurred due to Guardian rename operations on the SQL/MX objects. You must perform corresponding rename operations on the backup system to synchronize primary and backup systems. You must ensure that consistency is maintained between the primary and backup environments when Guardian rename operation is performed.

Note. You must stop the RDF updaters to perform Guardian rename on the backup system. After the rename operation is complete, restart the RDF updaters.

MODIFY and Redefinition Timestamp

The Guardian rename operation changes the redefinition timestamp of the object but it is the same physical object with a different Guardian name. Therefore, after Guardian rename operation is performed, the similarity check will pass. The following example illustrates this occurrence:

```
>>prepare q from select * from t1;
--- SQL command prepared.
>>modify table t1
+>rename location <existing location> to <new file name
part>;
--- SQL operation complete.
>>execute q;
*** WARNING[8578] Similarity check passed.
<successful result>
>>
```

In the example, although `execute q` uses the original name to access the renamed table t1, the similarity check passes because table t1 remains the same physical table. The Guardian rename of a base table changes the redefinition timestamp of the table but not indexes. However, the Guardian rename of an index changes the redefinition timestamp of the index, but not the base table and other indexes on that table.

MODIFY and Table Reloading

Some of MODIFY's options start a FUP RELOAD process that runs in the background. Until this process completes, you cannot do DDL or utility operations on the file. You can monitor the reload process's progress with this command:

FUP STATUS *physical-file-name*

If FUP STATUS returns a RELOAD COMPLETED message and the physical file is not being opened by another process, you can start the next MODIFY operation. Note that the ORSERV process started by the reload operation might still open the physical file a bit longer (about five minutes) even though FUP STATUS already returns the RELOAD COMPLETED message.

You can find the Guardian (physical) file name by using the SHOWLABEL command. For example:

```
SHOWLABEL CAT.SCH.T1, DETAIL;
```

For details on this command, see SHOWLABEL Command on page 4-99.

Correcting File Name Problems with MODIFY

When you create a table or index with precise file names, a file might already exist with the same name as one of the partitions to be created. Typically, the solution is to move the partition that already exists. Use MODIFY TABLE...MOVE PARTITION to specify a new Guardian file name for the partition to be moved. This file can exist on the same volume as the original or on a different volume.

For example, suppose that you need to recover this table:

```
CREATE TABLE T13B (....)
location $data2.ZSDXQGN5.P0000000
....
(add location $DATA2.ZSDXQGN5.P0010000,
add location $DATA2.ZSDXQGN5.P0020000,
add location $DATA2.ZSDXQGN5.P0030000);
```

Suppose that, however unlikely, a file with the name $DATA2.ZSDXQGN5.P0020000 already exists as a partition of another table, say T13x. Use MODIFY to move this partition of T13x:

```
MODIFY TABLE CAT.SCH.T13x MOVE PARTITION WHERE LOCATION
$DATA2.ZSDXQGN5.P0020000 TO LOCATION $DATA.ZSDXQGN5.P002A000;
```

Following this operation, you can properly perform the original CREATE TABLE statement with the indicated file names. You can also use this technique to correct other individual file name problems, such as errors in naming individual partitions in previous commands.

Examples of MODIFY

- Move all records of an existing range partition to a new location:

```
MODIFY TABLE tab1 MOVE PARTITION
     WHERE LOCATION $data02
     TO LOCATION $data03;
```

- Move records of an existing range partition, whose key is equal to 10000 to the last key, to a new location:

```
MODIFY TABLE tab1 MOVE PARTITION
     WHERE KEY = VALUE (10000) THRU KEY = LAST KEY
     TO LOCATION $data02
     EXTENT (512, 512) MAXEXTENTS 256;
     TO LOCATION $data03
     EXTENT (1024, 1024) MAXEXTENTS 256;
```

- Move the second partition of a hash partitioned table to a new location:

```
MODIFY TABLE tab1 MOVE PARTITION
     WHERE KEY = VALUE (2)
     TO LOCATION $DATA02;
```

- Move a partition of a hash partitioned table from $data02 to $data03:

```
MODIFY TABLE tab1 MOVE PARTITION
     WHERE LOCATION $data02
     TO LOCATION $data03
     EXTENT (1024, 1024) MAXEXTENTS 256;
```

- Modify table with an online operation

```
MODIFY TABLE MODT408A05 ADD PARTITION
      WHERE KEY= first key upto key= value (30000)
      TO LOCATION $data04
      WITH SHARED ACCESS
      COMMIT BEFORE '2007-04-05 16:25:40'
     ONCOMMITERROR COMMIT WORK AFTER '2007-04-05 25:19:00';
```

- The following example shows a simple rename of a single Guardian location of a table. Consider a Table T1 with location as `$DATA01.ZSDABCDE.SPARTN00.`

```
MODIFY TABLE CAT.SCH.T1 RENAME WHERE LOCATION
$DATA01.ZSDABCDE.SPARTN00 TO TPARTN00;
```

The location `SPARTN00` is renamed to `TPARTN00.`The volume and subvolume are not renamed.

- Following example shows rename of a set of specified locations of a table

```
MODIFY TABLE CAT.SCH.T2
  RENAME LOCATION
   ( $DATA01.ZSDABCDE.ABCDEF00 TO FEDCBA00
    , $DATA02.ZSDABCDE.GHIJKL00 TO LKJIHG00 );
```

The keyword `WHERE` is optional.

- The following example shows the rename of all locations of a table matching a pattern. Consider a table T1 with locations $DATA01.ZSDABCDE.SPARTX00 and $DATA02.ZSDABCDE.SPARTY00.

```
MODIFY TABLE CAT.SCH.T1
    RENAME WHERE LOCATION $*.ZSDABCDE.S* MAP NAMES TO
T???????;
```

Both locations match the pattern and therefore, are renamed. The resulting locations contains the letter ‘T’ in the leftmost position, and the letters in the remaining positions are same as the original location. Therefore, locations will be renamed $DATA01.ZSDABCDE.TPARTX00 and $DATA02.ZSDABCDE.TPARTY00.

- Renaming a set of specified locations of an index

```
MODIFY INDEX CAT.SCH.IDX2
       RENAME LOCATION
        ( $DATA01.ZSDABCDE.ABCDEF00 TO FEDCBA00
        , $DATA02.ZSDABCDE.GHIJKL00 TO LKJIHG00 );
```

- Renaming all locations matching a pattern of a filename of an index

```
MODIFY INDEX CAT.SCH.MYIX
     RENAME WHERE LOCATION $DATA*.ZSDXYZZY.S* TO T?????00;
```

In the example, the mapping specification `T?????00` indicates that the initial character of the target file name must be 'T', regardless of the character in the source file name. The subsequent five characters will be copied from the corresponding positions in the source file name and the last two will be `00`.

- This example moves the partition of a sequence generator:

```
MODIFY SEQUENCE seq1 MOVE PARTITION TO LOCATION $data02;
```

- This example renames the Guardian location of a sequence generator:

```
MODIFY SEQUENCE seq1 RENAME WHERE LOCATION $DATA06.ZSD
GQJL7.GGLP9700 to NEWLOC00;
```

- This example uses a pattern to rename the Guardian location of a sequence generator:

```
MODIFY SEQUENCE seq1 RENAME WHERE LOCATION $*.*.* MAP NAMES
TO NW????00;
```

- Renaming a single Guardian location of an index

```
MODIFY INDEX CAT.SCH.IDX1
RENAME WHERE LOCATION
$DATA01.ZSDABCDE.SPARTN00 TO TPARTN00;
```

Note that the keyword `WHERE` is optional.

PREPARE Statement

The PREPARE statement compiles a dynamic SQL statement for later execution with the EXECUTE statement. You can use PREPARE in an MXCI session or in an embedded SQL program.

C/COBOL The application must supply a name to be associated with the prepared statement. ■

MXCI You can also use PREPARE to check the syntax of a statement without executing the statement in MXCI. ■

MXCI

```
PREPARE statement-name FROM statement ■
```

C/COBOL

```
PREPARE statement-name FROM SQL-statement-variable

SQL-statement-name is:
   statement-name | ext-statement-name

ext-statement-name is:
   [GLOBAL | LOCAL] value-specification ■
```

statement-name

is an SQL identifier that specifies a name to be used for the prepared statement. See Identifiers on page 6-56. If you specify the name of an existing prepared statement, the new statement overwrites the previous one.

In MXCI, the statement name is not case-sensitive unless you delimit it within double quotes.

In embedded SQL, the statement name is case-sensitive. For example, the statement named `findemp` is not equivalent to the statement named `FINDEMP`.

MXCI *statement*

specifies the SQL statement to prepare. ■

C/COBOL *ext-statement-name*

is a *value-specification*—a host variable with character data type (for example, `:stmt`). When PREPARE executes, the content of the value specification identifies the statement that can be executed at a later time with EXECUTE. If the statement name corresponds to the name of a previously prepared statement, the new prepared statement overwrites the previous one. ■

C/COBOL `GLOBAL | LOCAL`

specifies the scope of the prepared statement. The default is LOCAL. A GLOBAL prepared statement can be executed within the SQL session. A LOCAL prepared statement can be executed only within the module or compilation unit in which it was prepared. ■

C/COBOL *`SQL-statement-variable`*

is a *`value-specification`*—a host variable with character data type that specifies the SQL statement to prepare (for example, `:stmt_buffer`). ■

Considerations for PREPARE

You cannot PREPARE a compound statement.

Availability of a Prepared Statement

MXCI

If a PREPARE statement fails, any subsequent attempt to execute the named statement fails.

Only the MXCI session that executes the PREPARE can execute the associated prepared statement. The prepared statement is available for execution until the MXCI session terminates or until it executes another PREPARE statement that uses the same statement name (either successfully or unsuccessfully).

A statement must be compiled by PREPARE before you can EXECUTE it, but after it is compiled, you can EXECUTE the statement multiple times without recompiling it. In particular, you can execute a prepared statement multiple times with different parameter values. See MXCI Examples of EXECUTE on page 2-204. ■

Dynamic Parameters

C/COBOL

A preparable statement can include any number of dynamic parameter markers (or specifications). The syntax for a dynamic parameter marker is a question mark (?). The data types of these markers are inferred from the context of the SQL statement. ■

Identifying Statements

C/COBOL

Each SQL/MX statement embedded in a program or compiled with PREPARE has a statement name that is an SQL identifier. If you compile a dynamic SQL statement with PREPARE, you can specify a name for the SQL statement in the PREPARE statement. For example:

```
EXEC SQL PREPARE ins_cust FROM :stmt_buffer;
```

where `ins_cust` is the name of the SQL statement.

To name a static SQL statement explicitly, precede the statement with a comment. For more information, see C/C++ and COBOL comments in the *SQL/MX Programming Manual for C and COBOL*.

If you do not specify a name for a static SQL statement, the SQL 3GL preprocessor assigns the statement a name of the form SQLMX_DEFAULT_STATEMENT_*n*, where *n* is an integer incremented by the preprocessor. ■

C/COBOL

Statement Names

You cannot have more than one statement allocated with the same name within the same scope. For example, this sequence from a C program is not valid:

```
strcpy(stmt1,"STMT1");
EXEC SQL PREPARE :stmt1 FROM :stmt_buffer1;
...
strcpy(stmt2,"STMT1");
EXEC SQL PREPARE :stmt2 FROM :stmt_buffer2;
```

The second PREPARE fails because STMT1 has already been prepared. ■

MXCI Examples of PREPARE

- Prepare a SELECT statement, naming it FINDEMP, and then execute FINDEMP:

```
PREPARE FINDEMP FROM
  SELECT * FROM persnl.employee
  WHERE salary > 40000.00 AND jobcode = 450;
--- SQL command prepared.

EXECUTE findemp;

EMPNUM  FIRST_NAME  LAST_NAME   DEPTNUM  JOBCODE  SALARY
------  ----------  ----------  -------  -------  --------
   232  THOMAS      SPINNER        4000      450  45000.00

--- 1 row(s) selected.
```

- Prepare a SELECT statement, naming it EMPCOM, and then enter the DISPLAY STATISTICS command to display the preparation statistics:

```
PREPARE EMPCOM FROM
  SELECT first_name, last_name, deptnum
  FROM persnl.employee
  WHERE deptnum <> 1500
    AND salary <= (SELECT AVG (salary)
                   FROM persnl.employee
                   WHERE deptnum = 1500);
--- SQL command prepared.

DISPLAY STATISTICS;

Start Time             2000/04/24 15:09:39.439
End Time               2000/04/24 15:09:46.946
Elapsed Time                      00:00:07.507
Compile Time                      00:00:07.507
Execution Time                    00:00:00.000
```

C Examples of PREPARE

- Prepare and execute an INSERT statement:

```
...
strcpy(stmt_buffer,"INSERT INTO SALES.CUSTOMER"
  " (CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE)"
  " VALUES (1120, 'EXPERT MAILERS', '5769 N.25TH PLACE',"
  " 'PHOENIX', 'ARIZONA', '85016')");
...
EXEC SQL PREPARE ins_cust FROM :stmt_buffer;
...
EXEC SQL EXECUTE ins_cust;
...
```

- Prepare and execute an UPDATE statement with dynamic input parameters:

```
...
strcpy(stmt_buffer,"UPDATE SALES.CUSTOMER SET CREDIT = ?"
  " WHERE CUSTNUM = CAST(? AS NUMERIC(4) UNSIGNED)");
...
EXEC SQL PREPARE upd_cust FROM :stmt_buffer;
...
/* Input values for parameters into host variables */
scanf("%s",in_credit);
scanf("%ld",&in_custnum);
...
EXEC SQL EXECUTE upd_cust USING :in_credit, :in_custnum;
...
```

- This example uses extended statement names:

```
...
strcpy(stmt,"ins_cust1");
EXEC SQL PREPARE :stmt FROM :stmt_buffer;
EXEC SQL EXECUTE :stmt;
...
strcpy(stmt,"ins_cust2");
EXEC SQL PREPARE :stmt FROM :stmt_buffer;
EXEC SQL EXECUTE :stmt;
```

COBOL Examples of PREPARE

- Prepare and execute an INSERT statement:

```
          ...
          MOVE "INSERT INTO SALES.CUSTOMER
     -     " (CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE)
     -     " VALUES (1120, 'EXPERT MAILERS', '5769 N.25TH PLACE',
     -     " 'PHOENIX', 'ARIZONA', '85016')"
          TO stmt-buffer.
          ...
          EXEC SQL PREPARE ins_cust FROM :stmt-buffer END-EXEC.
          ...
          EXEC SQL EXECUTE ins_cust END-EXEC.
          ...
```

- Prepare and execute an UPDATE statement with dynamic input parameters:

```
          ...
          MOVE "UPDATE SALES.CUSTOMER SET CREDIT = ? "
             & " WHERE CUSTNUM = CAST(? AS NUMERIC(4) UNSIGNED)")"
          TO stmt-buffer.
          ...
          EXEC SQL PREPARE upd_cust FROM :stmt-buffer END-EXEC.
          ...
     * Input values for parameters into host variables
          ACCEPT in-credit.
          ACCEPT in-custnum.
          ...
          EXEC SQL EXECUTE upd_cust
                   USING :in-credit, :in-custnum
          END-EXEC.
          ...
```

- This example uses extended statement names:

```
          ...
          MOVE "ins_cust1" TO stmt.
          EXEC SQL PREPARE :stmt FROM :stmt-buffer END-EXEC.
          EXEC SQL EXECUTE :stmt END-EXEC.
          ...
          MOVE "ins_cust2" TO stmt.
          EXEC SQL PREPARE :stmt FROM :stmt-buffer END-EXEC.
          EXEC SQL EXECUTE :stmt END-EXEC.
```

POPULATE INDEX Utility

Considerations for POPULATE INDEX
Examples of POPULATE INDEX

POPULATE INDEX records the status in the DDL_LOCKS metadata table. You can query this table to determine the progress of POPULATE INDEX operation.POPULATE INDEX is a syntax-based utility that can be executed through MXCI. The POPULATE INDEX utility loads SQL/MX indexes.

```
POPULATE INDEX index-name ON table-name [option]

index-name ::= name

table-name ::= [[catalog.]schema.]name

option ::= with-shared-access

with-shared-access ::= WITH SHARED ACCESS [commit-options]

commit-options ::=

    {COMMIT   [WORK] [    WHEN READY      ] [on-error] }
    {                [ {  AFTER time   } ]             }
    {                [ {  BEFORE time  } ]             }
    {ROLLBACK [WORK]                                     }

on-error::= [ ONCOMMITERROR commit-options ]

time::= yyyy-mm-dd hr:min:sec[.precision]
```

Syntax Description of POPULATE INDEX

`index-name`

is an SQL identifier that specifies the simple name for the index. You cannot qualify *index-name* with its schema name. Indexes have their own namespace within a schema, so an index name might be the same as a table or constraint name. However, no two indexes in a schema can have the same name.

`table-name`

is the name of the *table-name* for which to populate the index. See Database Object Names on page 6-13.

`WITH SHARED ACCESS`

specifies that the operation is an online operation. If *commit-options* is omitted, the effect is the same as specifying COMMIT WHEN READY ONCOMMITERROR ROLLBACK WORK.

```
COMMIT [WORK] [  WHEN READY        ]  [on-error]
              [ {  AFTER time  } ]
              [ {  BEFORE time } ]
```

specifies the time at which the Commit Phase should occur. COMMIT WHEN READY specifies that the Commit Phase should occur at the earliest possible time. COMMIT AFTER *time* specifies that the Commit Phase should occur after the given *time*. COMMIT BEFORE *time* specifies that the Commit Phase should occur before the given *time*. The *on-error* clause specifies what should happen if the Commit Phase fails with a retryable error. If omitted, the effect is the same as specifying ONCOMMITERROR ROLLBACK WORK. The *time* is a Datetime value.

`ROLLBACK WORK`

specifies that the operation should be terminated. ROLLBACK WORK is specified in an *on-error* clause.

`ONCOMMITERROR` *commit-options*

specifies what action SQL should take if a retryable error occurs during the commit phase. Retryable errors include file in use, lock request time-outs, resource unavailability, and missing a BEFORE or AFTER time window.

A non-retryable error always causes SQL to cancel changes to the database and terminates the operation, no matter what you specify in the ONCOMMITERROR option. ONCOMMITERROR is recursive because it appears within a COMMIT option and specifies another COMMIT option. You can specify up to three COMMIT options on a single statement. If you do not specify the ONCOMMITERROR clause, the effect is the same as specifying ONCOMMITERROR ROLLBACK WORK.

Considerations for POPULATE INDEX

- When POPULATE INDEX is being executed with shared access, the base table is accessible for read write DML operations during that time period, except during the commit phase at the end.
- When POPULATE INDEX is being executed without shared access base table is available for READ DML operations only till populate index completes.
- During an online POPULATE INDEX utility request, you can control when to commit the transaction by giving a specific time. Specify an AFTER clause or a BEFORE clause to control when to make the index available.
- Retryable errors are errors that occur during any phase of the execution of an online POPULATE INDEX. Retryable errors include file in use, lock request time-outs, resource unavailability, and missing a BEFORE or AFTER time window. Some errors are retried automatically. Errors detected during the COMMIT phase are retried based on the ONCOMMITERROR clause. Errors detected during the

initial and the data movement phases are retried individually and may include retry counts.

- Errors can occur if the source base table or target index cannot be accessed, or if the load fails due to some resource problem or problem in the file system.
- POPULATE INDEX does not work with SQL/MP alias names.
- POPULATE INDEX need sufficient memory to copy the table and to create the index respectively.
- POPULATE INDEX records the status in the DDL_LOCKS metadata table. You can query this table to determine the progress of POPULATE INDEX operation.

Status	**Description**
1	The DDL Lock row is created.
3	File Labels of index are updated.
4	Load of index has started.
5	Load of index has completed.
7	Index is online. POPULATE INDEX operation has completed.

Examples of POPULATE INDEX

- This example loads the specified index from the specified table:

```
POPULATE INDEX myindex ON mycat.myschema.mytable;
```

- This example loads the specified index from the specified table, which uses the default catalog and schema:

```
POPULATE INDEX index2 ON table2;
```

- This is an example for online operation which loads the specified index from the specified table, and uses the default catalog and schema:

```
POPULATE INDEX index2 ON table2 with-shared-access;
        COMMIT BEFORE '2007-04-05 16:25:40'
 ONCOMMITERROR COMMIT WORK AFTER '2007-04-05 25:19:00';
```

PURGEDATA Utility

Considerations for PURGEDATA
Examples of PURGEDATA

PURGEDATA is a syntax-based utility that can be executed through MXCI. The PURGEDATA utility deletes all data from an SQL/MX table and its related indexes or from specified partitions of a table that has no indexes.

```
PURGEDATA table-name [list-of-partitions] [IGNORE_TRIGGER]

table-name is:
   [ [catalog.]schema.]name

list-of-partitions is:
   [PARTITION] WHERE partition-map

partition-map is:
   {guardian-name | first-key }

guardian-name is:
   LOCATION [\node].$volume[.subvolume.file]

first-key is:
   {[KEY=] partition-id
   |[KEY=] partition-id UPTO [KEY=] partition-id
   |[KEY=] partition-id THRU [KEY=] partition-id }

partition-id is:
   {{FIRST | LAST} PARTITION
   | key-value }

key-value is:
  VALUE (column-value [, column-value] ...)
```

Syntax Description of PURGEDATA

table-name

is the name of the table from which the data is to be deleted. You can specify delimited and regular identifiers. If you do not specify the catalog and schema parts of the *table-name*, the default catalog and schema values for that session are applied. PURGEDATA returns errors if the catalog name does not exist, if the schema name is invalid, if the table name does not exist, or if the object name specifies an invalid object such as a view or an SQL/MP alias.

list-of-partitions

is the optional clause that specifies a subset of partitions to purge. If you do not specify this clause, PURGEDATA purges data from all partitions and dependent indexes. If you specify this clause, PURGEDATA purges data from the partitions in this list. PURGEDATA returns errors if the specified partitions do not exist, if dependent indexes exist on the table, and if the source object does not exist.

partition-map

describes a partition or range of partitions:

guardian-name

specifies the partially or fully qualified Guardian name that identifies the partition. If you specify only the volume, all partitions on the volume that belong to the object are affected. If you do not specify *\node*, only the local system is assumed. PURGEDATA returns an error if *guardian-name* is not valid.

\node can refer to a remote system that must be visible to the current (local) system.

first-key

identifies the partition by its first key value. Within the current object, every partition is assigned a unique first key value. You can use this value later to identify the partition. An error is returned if the first key value cannot be found. *first-key* must match an existing key value for PURGEDATA requests. Partition management operations can use different values for boundary operations.

FIRST

specifies the first partition. For range-partitioned objects, the first partition contains low or high values, depending on the ASC/DESC attribute. For hash-partitioned objects, the first partition is zero (0).

LAST

specifies the last partition. For range-partitioned objects, the last partition is the last value where its first key is greater than other first keys. For hash-partitioned objects, the last partition is the maximum number of partitions available minus one (1).

key-value

specifies the first key value used when the partition was created. You can specify a range of partitions.

IGNORE_TRIGGER

specifies that PURGEDATA should ignore DELETE triggers on the table. If they are not ignored and a DELETE trigger exists, PURGEDATA fails.

Considerations for PURGEDATA

- An error is returned if you specify a *list-of-partitions* for a hash-partitioned table. For hash-partitioned objects, data must be purged from the entire table.
- Errors are returned if *table-name* cannot be accessed or if a resource or file system problem causes the delete to fail.
- If PURGEDATA fails in response to a process, CPU, or system error, you must run the RECOVER utility to recover the operation. If the PURGEDATA operation cannot be canceled, RECOVER returns an error. See Checking DDL Locks on page 2-9 for details.
- PURGEDATA records operation progress steps in the DDL_LOCKS metadata table. Users can query this table to determine the PURGEDATA operation's progress:

PURGEDATA Operation Step	Step Progress Status
Step 1	DDL lock has been created.
Step 2	PURGEDATA has passed verification tests.
Step 3	Affected partitions have been marked corrupt.
Step 4	Partition $*volume*.zsd*nnnnn*.*nnnnnn*00 has been purged.*
Step 5	PURGEDATA operation has deleted data from all requested partitions.
Step 6	Affected partitions are now available.
Step 7	DDL lock has been removed.

* After this step, when any partition has been purged, a rollback using the RECOVER utility is not possible.

- PURGEDATA returns an error if a user transaction exists.
- PURGEDATA returns an error if you attempt a PURGEDATA operation on an SQL/MX metadata table (histogram, system defaults, or MXCS metadata tables).
- PURGEDATA returns an error if a DELETE trigger is defined on *table-name,* unless the IGNORE_TRIGGER option is used.
- PURGEDATA returns an error if *table-name* is being referenced by one or more referential integrity constraints, unless all the referencing tables are empty.

  PURGEDATA sets the corrupt bit while processing. If PURGEDATA fails before it completes, the table and its dependent indexes will still be corrupt and you must run RECOVER with the RESUME option to complete the operation and remove the data. File system error 59 is returned if you try to access a table whose corrupt bit is set.

Examples of PURGEDATA

- This example purges the data in the specified table. If the table has indexes, their data is also purged.

```
PURGEDATA mycat.myschema.mytable;
```

- This example purges the data in the specified partition, which has a Guardian name:

```
PURGEDATA mycat.myschema.mytable
WHERE LOCATION $DATA1.ZSDA09TO.QZ780000;
```

- This example purges data from all partitions of the table:

```
PURGEDATA mycat.myschema.mytable
WHERE KEY = FIRST PARTITION THRU LAST PARTITION;
```

RECOVER Utility

Considerations for RECOVER
Examples of RECOVER

RECOVER is a syntax-based utility that can be executed through MXCI. The RECOVER utility determines the state of a failed utility operation and executes its recovery procedure. RECOVER completes the failed utility operation by rolling back the entire operation or by completing the operation. In most cases, RECOVER rolls back the utility operation by making the state the same as it was before the operation started.

```
RECOVER [INCOMPLETE SQLDDL OPERATION ON] object [ddl-lock] [opt]

object is:
    { TABLE | INDEX } object-name

object-name is:
    [[catalog.]schema.]name

ddl-lock is:
    WITH DDL_LOCKS ddl-lockname

ddl-lockname is:
    catalog.schema.name

opt is:
    { CANCEL | RESUME }
```

Syntax Description of RECOVER

object

is the name of the table or index that needs to be recovered. It must be the same object that you were attempting to update with the command that failed. You can specify delimited or regular identifiers. If you do not specify the catalog or schema parts of *object-name*, RECOVER uses the default catalog and schema values for that session. RECOVER returns errors if the catalog name is invalid, if the schema name is invalid, if the table name is invalid, or if *object-name* does not need to be recovered.

ddl-lock

is the fully qualified name of the DDL_LOCKS object that was created by the utility operation that failed. *ddl-lock* is optional because there is only one lock allowed on a object at a time. RECOVER can determine the *ddl-lock* name from the object.

opt

directs how RECOVER should proceed with the operation.

`CANCEL`

If you specify `CANCEL`, RECOVER attempts to undo the effects of the failed utility operation. Otherwise, recovery fails. The default is `CANCEL`.

`RESUME`

If you specify `RESUME`, RECOVER attempts to carry the failed utility operation to its completion. Otherwise, the recovery fails.

Considerations for RECOVER

- *`object`* must be the same object that you were attempting to update with the command that failed. Suppose that you have a table, cat.sch.t that has an index cat.sch.i. You performed a POPULATE INDEX command for that index but it failed. Consider these RECOVER operations:

```
>>recover table cat.sch.t;

*** ERROR[20209] Nothing remains to be recovered on
CAT.SCH.T.

--- SQL operation failed with errors.
>>recover index cat.sch.i;

--- SQL operation complete.
>>
```

- At the completion of a RECOVER operation, the original failed operation should either be completed as specified (`RESUME`) or completely rolled back (`CANCEL`).
- Different utilities can be canceled or resumed depending on their failure status.
- See the description of the individual utility operation to determine when and if `CANCEL` or `RESUME` should be specified.

Examples of RECOVER

- This example recovers and by default cancels the incomplete SQL DDL operation on the specified table:

```
RECOVER TABLE mycat.myschema.mytable;
```

- This example explicitly instructs RECOVER to cancel the failed utility operation:

```
RECOVER TABLE mycat.myschema.mytable CANCEL;
```

RECOVER SCHEMA Operation

Considerations for RECOVER SCHEMA
Examples of RECOVER SCHEMA

The RECOVER SCHEMA operation is used to revert the changes made by a failed change ownership operation or to carry it through to completion.

```
RECOVER SCHEMA schema [CANCEL|RESUME]
```

schema

specifies the name of the schema on which the failed change ownership operation will be recovered.

[CANCEL]

The RECOVER SCHEMA operation attempts to undo the effects of the failed change ownership operation. The default option is CANCEL.

[RESUME]

The RECOVER SCHEMA operation attempts to carry the failed change ownership operation through to completion.

Considerations for RECOVER SCHEMA

Authorization and Availability Requirements

The RECOVER SCHEMA operation can be performed by the original schema owner, by a security administrator, or by the Super ID (if Super ID is part of the Security Administrator's group or if no Security Administrator's group exists). After successful completion of this operation, existing opens on the underlying objects are invalidated.

DDL Locks

The RECOVER SCHEMA operation checks for DDL locks acquired with the GIVE SCHEMA operation on a particular schema. If no DDL lock is found, RECOVER SCHEMA fails. If a DDL lock is found, the RECOVER SCHEMA undoes or completes the failed GIVE SCHEMA operation and releases the lock.

Examples of RECOVER SCHEMA

- This example identifies the schema affected by a failed change ownership operation and instructs RECOVER SCHEMA to cancel the failed change ownership operation:

  ```
  >>recover schema userschema2 cancel;
  --- SQL operation complete.
  ```

  OR

```
>>recover schema userschema2;
--- SQL operation complete.
```

- This example identifies the schema affected by a failed change ownership operation and instructs RECOVER SCHEMA to resume the failed change ownership operation:

```
>>recover schema userschema2 resume;
--- SQL operation complete.
```

REGISTER CATALOG Statement

The REGISTER CATALOG statement registers an SQL/MX catalog on a remote node. A catalog is not visible to a remote node until you register it.

```
REGISTER CATALOG catalog ON \node.$volume [RESTRICT |
   CASCADE]
```

catalog

is the ANSI name of the target catalog. It must be visible on the local node. No catalog with the same name can exist on the target node. You cannot register the system catalog.

node.$volume

is the remote node on which the catalog will be registered.

RESTRICT

specifies that only the named catalog is registered. If that catalog is related to other catalogs, an error occurs.

RESTRICT is the default.

CASCADE

specifies that the named catalog, and any catalogs that are directly or indirectly related to it, will be registered.

Considerations for REGISTER CATALOG

REGISTER CATALOG creates an empty catalog reference on the target node and updates automatic catalog references. If a catalog reference already exists on the target node with a different volume name and no definition schemas exist on that node for that catalog, NonStop SQL/MX changes the volume name to *volume*.

Authorization and Availability Requirements

To register a catalog, you must be the user who created the catalog or be the super ID.

A transaction can be user-initiated or system-initiated. If no transaction exists, NonStop SQL/MX automatically starts one.

Examples of REGISTER CATALOG

- This command registers a catalog on another node:

  ```
  REGISTER CATALOG mycat ON \nodex.$data47;
  ```

- Suppose that you create a view that references two tables in different catalogs:

  ```
  CREATE VIEW view_catalog.view_schema.MYVIEW4
    (v_ordernum, v_partnum) AS
    SELECT od.ordernum, p.partnum
      FROM SALES.SALES_SCHEMA.ODETAIL OD
          INNER JOIN CUST.CUST_SCHEMA.ORDERS P
      ON od.partnum = p.partnum;
  ```

  If you issue either of these commands:

  ```
  REGISTER CATALOG SALES ON \nodex.$data47;
  REGISTER CATALOG SALES ON \nodex.$data47 RESTRICT;
  ```

  NonStop SQL/MX returns an error because catalog SALES is related to catalog CUST by this view, and view_catalog is related to both. The command defaults to the RESTRICT option.

  If you issue this command:

  ```
  REGISTER CATALOG SALES ON \nodex.$data47 CASCADE;
  ```

  NonStop SQL/MX registers all three catalogs, SALES, CUST, and view_catalog, on \nodex because they are related to each other by this view.

- Suppose that you create two views that are indirectly related to each other:

  ```
  CREATE VIEW cat1.sch.v AS
    SELECT * FROM cat2.sch.t;

  CREATE VIEW cat2.sch.v AS
    SELECT * FROM cat3.sch.t;
  ```

  In this example, cat1 and cat3 are each directly related to cat2, and each is indirectly related to the other. You must register all three catalogs (or one of them withe the CASCADE option) so that their tables will be visible on another node.

REVOKE Statement

[Considerations for REVOKE](#)
[Examples of REVOKE](#)

The REVOKE statement revokes access privileges for an SQL/MX table, view, sequence generator, or stored procedure from specified users. For more information, see REVOKE EXECUTE Statement on page 2-323.

```
REVOKE [GRANT OPTION FOR]
   {privilege [,privilege ]... | ALL [PRIVILEGES]}
   ON [TABLE | SEQUENCE ] object
   FROM {grantee [,grantee]...} [drop-behavior]
   [BY authid-grantor]

grantee is:
   authid | PUBLIC

privilege is:
     SELECT
   | DELETE
   | INSERT
   | UPDATE [(column [,column]...)]
   | REFERENCES [(column [,column]...)]
   | USAGE

drop-behavior is:

CASCADE | RESTRICT
```

Syntax Description of REVOKE

`GRANT OPTION FOR`

specifies that the WITH GRANT OPTION for the privilege must be revoked. The privilege itself is not revoked.

`privilege [,privilege]... | ALL [PRIVILEGES]`

specifies the privileges to revoke, as follows. You can specify each of these privileges for a table or view. For a stored procedure, ALL PRIVILEGES revokes only the EXECUTE privilege. For a sequence generator, ALL PRIVILEGES revokes only the USAGE privilege. See also REVOKE EXECUTE Statement on page 2-323.

SELECT	Can use SELECT statement.
DELETE	Can use DELETE statement.
INSERT	Can use INSERT statement.
UPDATE	Can use UPDATE statement.

REFERENCES	Can create constraints that reference the object.
USAGE	Can use the pseudocolumns, CURRVAL and NEXTVAL to access the sequence generator values.
ALL PRIVILEGES	Can have all privileges that apply to the object type.

`(`*`column`* `[,`*`column`*`]...)`

names the columns of the object to which the UPDATE or REFERENCES privilege applies. If you specify UPDATE or REFERENCES without column names, the privileges apply to all columns of the table or view.

`ON [TABLE | SEQUENCE]` *`object`*

specifies a table, sequence generator or view on which to revoke privileges. When the object is a stored procedure, the only privilege that you can specify is ALL PRIVILEGES. See REVOKE EXECUTE Statement on page 2-323 to revoke privileges for stored procedures.

`FROM {`*`authid`* `[,`*`authid`* `]... | PUBLIC}`

specifies one or more users from whom you revoke privileges.

`authid` specifies an authorization ID from whom you revoke privileges. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name (for example, `sql.user1`) or Guardian user number(for example, `255,255`). *`authid`* is not case-sensitive.

SQL:1999 specifies two special authorization IDs: PUBLIC and SYSTEM.

- PUBLIC specifies all present and future authorization IDs.
- SYSTEM specifies the implicit grantor of privileges to the creators of objects.

You cannot specify SYSTEM as an *`authid`* in a REVOKE statement.

`drop-behavior`

If you specify RESTRICT, the REVOKE operation fails if there are privilege descriptors or objects that would no longer be valid after the specified privileges are removed.

If you specify CASCADE, any such dependent privilege descriptors and objects are removed as part of the REVOKE operation.

The default is RESTRICT.

`BY` *`authid-grantor`*

specifies the authorization ID *`authid-grantor`* on whose behalf the revoke operation is performed. Only the Super ID or a Security Administrator can use the BY clause unless the Security Administrators Group is not empty and the Super ID has not been designated as a Security Administrator in which case the Super ID is denied the use of this feature. The effect of using the BY clause is the same as if the *authid-grantor* were to issue the REVOKE directly (without using the BY clause). *authid-grantor* cannot be SYSTEM. If the Security Administrator's Group is empty, then *authid-grantor* must be a valid authorization ID and hold the privilege(s) being granted WITH GRANT OPTION. However, Security Administrators have Super REVOKE BY capabilities in which *authid-grantor* may be any valid authorization ID that previously granted the privileges on the target object.

Considerations for REVOKE

Authorization Requirements

Unless you are a Security Administrator or the Super ID, you can revoke only those privileges that you have previously granted to the user.

If you are a Security Administrator, then you are exempt from the above restriction. In addition, successfully revoking a privilege will revoke all instances of that privilege granted by any Security Administrators. It will not affect any owner-derived grants of that privilege (that is the owner-derived hierarchy of grants is not affected). However, Security Administrators can revoke owner-derived grants using REVOKE BY *authid-grantor*. Since Security Administrators may be the target of an owner-derived grant, they may hold any privilege derived WGO, in which case they may revoke that privilege like any ordinary user.

If you are the Super ID, then your revoke privileges depend on the Security Administrator's Group. If the Security Administrator's Group is empty, then you may revoke any privilege on any object. Such revokes behave like a REVOKE BY *authid-grantor* where the *authid-grantor* is the object owner.

If the Super ID is designated as a Security Administrator, then the Super ID has the same REVOKE privileges as any other Security Administrator. However, in this case, if BY *authid-grantor* is omitted, then the implied grantor is the Super ID instead of the object owner and successfully revoking a privilege will revoke all instances of that privilege granted by any Security Administrators (that is the Super ID behaves like any other Security Administrator ID). It will not affect any owner-derived grants of that privilege (that is the owner-derived hierarchy of grants is not affected).

If the Security Administrator's Group is not empty and the Super ID is not designated as a Security Administrator, the Super ID will have the same restrictions as any ordinary user with respect to the REVOKE statement.

If one or more of the privileges being revoked does not exist, the system returns a warning.

You can revoke the USAGE privilege on a sequence generator only if you have previously granted it to the user. To revoke privileges for a sequence generator by using the CASCADE option, you must own the sequence generator or be the Super ID or Security Administrator. If the Super ID has issued a REVOKE USAGE using the *BY authid-grantor* clause, the *authid-grantor* must have previously granted the

USAGE privilege to the specified authorization IDs. You cannot revoke privileges from a user of the system if you have granted privileges to PUBLIC.

Examples of REVOKE

- This example revokes one user’s SELECT privileges on a table:

```
REVOKE SELECT ON TABLE persnl.employee
  FROM "sql.user1" RESTRICT;
```

- This example revokes the privileges of granting SELECT and DELETE privileges on a table from two users:

```
REVOKE GRANT OPTION FOR SELECT, DELETE
  ON TABLE sales.odetail FROM "sql.user1", "sql.user2";
```

- This example revokes UPDATE privileges on two columns of a table:

```
REVOKE UPDATE (start_date, ship_timestamp)
  ON TABLE persnl.project FROM PUBLIC RESTRICT;
```

- In this example the super ID revokes SELECT and DELETE privileges on a table on behalf of `sql.user1`:

```
REVOKE SELECT, DELETE ON TABLE sales.odetail
   FROM "sql.user3" BY "sql.user1";
```

- This example revokes USAGE privilege on a sequence generator from `sql.user2`:

```
REVOKE ALL ON SEQUENCE myseq FROM "sql.user2";
```

REVOKE CREATE CATALOG Statement

Considerations for REVOKE CREATE CATALOG
Examples for REVOKE CREATE CATALOG

The REVOKE CREATE CATALOG revokes the privilege to create a catalog from specified users.

```
REVOKE CREATE CATALOG FROM {"grantee" [,"grantee"]...}
```

`{"`*`grantee`*`" [,"`*`grantee`*`"]...}`

are Guardian users from whom the catalog creation privilege will be revoked.

Considerations for REVOKE CREATE CATALOG

When CREATE CATALOG privilege is revoked from the last authorized user, any user can create a catalog.

Authorization and Availability Requirements

A security administrator or the Super ID (if Super ID is part of the Security Administrator's group or if no Security Administrator's group exists) can revoke the privilege to create a catalog granted to specified users.

CREATE CATALOG privilege cannot be revoked from PUBLIC.

Examples for REVOKE CREATE CATALOG

- This example revokes the privilege to create a catalog from users, "SQL.USER1", "SQL.USER2", and "SQL.USER3":

```
REVOKE CREATE CATALOG FROM "SQL.USER1";
REVOKE CREATE CATALOG FROM "SQL.USER2","SQL.USER3";
```

REVOKE CREATE SCHEMA Statement

Considerations for REVOKE CREATE SCHEMA
Example for REVOKE CREATE SCHEMA

The REVOKE CREATE SCHEMA revokes the privilege to create a schema in a catalog from specified users.

```
REVOKE CREATE SCHEMA ON catalog FROM {"grantee"
[,"grantee"]...}
```

catalog

is the name of the catalog in which the schema creation privilege will be revoked.

{"*grantee*" [,"*grantee*"]...}

are Guardian users from whom the schema creation privilege on the *catalog* will be revoked.

Considerations for REVOKE CREATE SCHEMA

When CREATE SCHEMA privilege is revoked from the last authorized user, any user can create a schema in the target catalog.

Authorization and Availability Requirements

A security administrator or catalog owner or the Super ID (if Super ID is part of the Security Administrator's group or if no Security Administrator's group exists) can revoke the privilege granted to specified users from creating a schema within the *catalog*.

CREATE SCHEMA privilege cannot be revoked from PUBLIC.

Example for REVOKE CREATE SCHEMA

- This example revokes the privilege to create a schema from users, “SQL.USER1” and “SQL.USER2”:

  ```
  REVOKE CREATE SCHEMA ON CAT FROM "SQL.USER1", "SQL.USER2";
  ```

REVOKE EXECUTE Statement

Considerations for REVOKE EXECUTE
Examples of REVOKE EXECUTE

The REVOKE EXECUTE statement removes privileges for calling a stored procedure in Java (SPJ) from one or more specified users.

```
REVOKE [GRANT OPTION FOR]
   EXECUTE
   ON [PROCEDURE] procedure-ref
   FROM {grantee [,grantee]...} [RESTRICT | CASCADE]
   [BY authid-grantor]

procedure-ref is:
   [[catalog-name.]schema-name.]procedure-name

grantee is:
   authid | PUBLIC
```

`GRANT OPTION FOR`

specifies that the WITH GRANT OPTION for the EXECUTE privilege is to be revoked. The EXECUTE privilege itself is not revoked.

`EXECUTE`

specifies the privilege of calling the specified stored procedure.

`ON [PROCEDURE] procedure-ref`

specifies the ANSI logical name of a stored procedure on which to revoke privileges, of the form:

`[[catalog-name.]schema-name.]procedure-name`

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see Identifiers on page 6-56.

`FROM {authid [,authid ]... | PUBLIC}`

specifies one or more users from whom you revoke privileges.

authid specifies an authorization ID from whom you revoke privileges. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name, enclosed in double quotes. A Guardian user number (for example, “255,255”) is not allowed. *authid* is not case-sensitive.

SQL:1999 specifies two special authorization IDs: PUBLIC and SYSTEM.

- PUBLIC specifies all present and future authorization IDs.

- SYSTEM specifies the implicit grantor of privileges to the creators of stored procedures.

You cannot specify SYSTEM as an *authid* in a REVOKE EXECUTE statement.

If you specify RESTRICT, the REVOKE operation fails if there are privilege descriptors that would no longer be valid after the EXECUTE privilege is removed.

If you specify CASCADE, any such dependent privilege descriptors are removed as part of the REVOKE EXECUTE operation. The default is RESTRICT.

BY *authid-grantor*

specifies the authorization ID *authid-grantor* on whose behalf the revoke operation is performed. The EXECUTE privilege being revoked must have been previously granted by *authid-grantor*. Only the super ID can use the BY clause. If another user attempts to do so, the system returns an error. The effect of using the BY clause is the same as if the *authid-grantor* were to issue the REVOKE EXECUTE statement directly (without using the BY clause).

authid-grantor must be a valid authorization ID and cannot be SYSTEM.

Considerations for REVOKE EXECUTE

Authorization and Availability Requirements

You can revoke EXECUTE privilege only if you have previously granted it to the user. If the privilege does not exist, the system returns a warning.

To revoke privileges by using the CASCADE option, you must own the stored procedure or be the super ID.

If the super ID has issued a REVOKE EXECUTE using the BY *authid-grantor* clause, the *authid-grantor* must have previously granted the EXECUTE privilege to the specified authorization IDs.

You cannot revoke privileges from a user of the system if you have granted privileges to PUBLIC.

Examples of REVOKE EXECUTE

- To revoke the WITH GRANT OPTION privilege on ADJUSTSALARY from user 'HR.BETTY,' the super ID issues this REVOKE EXECUTE statement:

  ```
  REVOKE GRANT OPTION FOR EXECUTE
    ON PROCEDURE samdbcat.persnl.adjustsalary
    FROM 'HR.BETTY'
    BY 'SYSMGT.BEN';
  ```

  The user 'HR.BETTY' no longer has the WITH GRANT OPTION privilege but still has EXECUTE privilege on ADJUSTSALARY.

- To revoke the EXECUTE privilege on ADJUSTSALARY from user 'HR.BETTY,' the super ID issues this REVOKE EXECUTE statement with the CASCADE option:

  ```
  REVOKE EXECUTE
    ON PROCEDURE samdbcat.persnl.adjustsalary
    FROM 'HR.BETTY' CASCADE
    BY 'SYSMGT.BEN';
  ```

- This REVOKE EXECUTE statement issued by the super ID fails because a dependent privilege exists for the users of the HR group to whom the user 'HR.BETTY' granted the EXECUTE privilege on ADJUSTSALARY:

  ```
  REVOKE EXECUTE
    ON PROCEDURE samdbcat.persnl.adjustsalary
    FROM 'HR.BETTY'
    BY 'SYSMGT.BEN';
  ```

  ```
  *** ERROR[1014] Privileges were not revoked.  Dependent
  privilege descriptors still exist.
  --- SQL operation failed with errors.
  ```

- This REVOKE EXECUTE statement issued by the super ID does not fail because no dependent privileges exist for the HR group, which had only EXECUTE privilege on ADJUSTSALARY:

  ```
  REVOKE EXECUTE
    ON PROCEDURE samdbcat.persnl.adjustsalary
    FROM 'HR.MIKE', 'HR.JOE', 'HR.HILDE' RESTRICT
    BY 'HR.BETTY';
  ```

REVOKE SECURITY_ADMIN Statement

[Considerations for REVOKE SECURITY_ADMIN]
[Examples of REVOKE SECURITY_ADMIN]

The REVOKE SECURITY_ADMIN statement removes the Security Administrator designation from a specified user.

```
REVOKE SECURITY_ADMIN from "grantee"
grantee is:
        authid | userid
```

authid

specifies the authorization ID from whom you are removing the Security Administrator designation. Authorization IDs identify users during the processing of SQL statements. The authorization ID must be a valid Guardian user name, enclosed in double quotes.

userid

specifies the authorization ID from whom you are removing the Security Administrator designation. The userid is any legal Guardian userid (group,user) enclosed in double quotes (for example, “255,255”). The userid is not required to be in the system USERID file. The userid option provides a means of revoking the Security Administrator designation from a user that has been deleted from the USERID file.

Considerations for REVOKE SECURITY_ADMIN

Authorization Requirements

If the Security Administrator's Group is empty, only the Super ID may execute the REVOKE SECURITY_ADMIN statement. Otherwise, only a Security Administrator may execute this statement.

Metadata Version Requirements

The REVOKE SECURITY_ADMIN statement requires system metadata version 3100 or greater. If the statement is executed with lower versions of the system metadata, a SQL error 25223 is generated.

Examples of REVOKE SECURITY_ADMIN

- The following example removes the Security Administrator designation from the user, SECADMIN.USER1:

```
REVOKE SECURITY_ADMIN FROM "SECADMIN.USER1";
```

- The following example removes the Security Administrator designation from the Super ID specified as a userid:

```
REVOKE SECURITY_ADMIN FROM "255,255";
```

ROLLBACK WORK Statement

The ROLLBACK WORK statement undoes all database modifications to audited objects made during the current transaction, releases all locks on audited objects held by the transaction, and ends the transaction. See Transaction Management on page 1-13.

```
ROLLBACK [WORK]
```

WORK is an optional keyword that has no effect.

ROLLBACK WORK has no effect if there is no active transaction.

Embed ROLLBACK WORK closes all open cursors in the application, because cursors do not span transaction boundaries. You cannot fetch with a cursor after a transaction ends without reopening it. ■

Considerations for ROLLBACK WORK

Begin and End a Transaction

BEGIN WORK starts a transaction. COMMIT WORK or ROLLBACK WORK ends a transaction.

MXCI Examples of ROLLBACK WORK

- Suppose that you add an order for two parts numbered 4130 to the ORDERS and ODETAIL tables. When you update the PARTLOC table to decrement the quantity available, you discover there is no such part number in the given location.

  Use ROLLBACK WORK to terminate the transaction without committing the database changes:

```
BEGIN WORK;

INSERT INTO sales.orders
  VALUES (124, DATE '1996-04-10',
          DATE '1996-06-10', 75, 7654);

INSERT INTO sales.odetail
  VALUES (124, 4130, 25000, 2);

UPDATE invent.partloc
  SET qty_on_hand = qty_on_hand - 2
  WHERE partnum = 4130 AND loc_code = 'K43';
```

```
ROLLBACK WORK;
```

ROLLBACK WORK cancels the inserts that occurred during the transaction and releases the locks held on ORDERS and ODETAIL.

C Examples of ROLLBACK WORK

- Start a transaction, execute an UPDATE statement, and test SQLSTATE. If the UPDATE is successful, the database changes are committed. Otherwise, the database changes are rolled back.

```
...
CHAR SQLSTATE_OK[6] = "00000";
EXEC SQL BEGIN DECLARE SECTION;
  CHAR SQLSTATE[6];
  ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL BEGIN WORK;            /* Start a transaction. */
...
EXEC SQL UPDATE ... ;           /* Change the database. */
...
if (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
   EXEC SQL COMMIT WORK;         /* Commit the changes. */
else
   EXEC SQL ROLLBACK WORK;    /* Roll back the changes. */
```

COBOL Examples of ROLLBACK WORK

- Start a transaction, execute an UPDATE statement, and test SQLSTATE. If the UPDATE is successful, the database changes are committed. Otherwise, the database changes are rolled back.

```
...
01 SQLSTATE_OK  PIC X(5) VALUE "00000".
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE     PIC X(5).
   ...
   EXEC SQL END DECLARE SECTION END-EXEC.
   ...
   EXEC SQL BEGIN WORK END-EXEC.
   ...
   EXEC SQL UPDATE ... END-EXEC.
   ...
   IF SQLSTATE = SQLSTATE_OK
      EXEC SQL COMMIT WORK END-EXEC.
   ELSE
      EXEC SQL ROLLBACK WORK END-EXEC.
   END-IF.
```

SELECT Statement

The SELECT statement is a DML statement that retrieves values from tables, views, derived tables determined by the evaluation of query expressions, or joined tables.

Embed The SELECT INTO statement is used to retrieve a single row of values from tables, views, joined tables, or derived tables determined by the evaluation of query expressions. It assigns the retrieved row of values to host variables. Use the INTO version of SELECT only in embedded SQL programs. ■

```
[ROWSET FOR size-and-index ]                                    (Embed)

size-and-index is:
| INPUT SIZE rowset-size-in
| OUTPUT SIZE rowset-size-out
| KEY BY row-id
| INPUT SIZE rowset-size-in, OUTPUT SIZE rowset-size-out
| INPUT SIZE rowset-size-in, KEY BY row-id
| OUTPUT SIZE rowset-size-out, KEY BY row-id
| INPUT SIZE rowset-size-in, OUTPUT SIZE rowset-size-out,
  KEY BY row-id

SELECT [[ANY N] | [FIRST N]] [ALL | DISTINCT] select-list

   INTO variable-spec [,variable-spec]... ■                     (Embed)

   FROM table-ref [,table-ref]...
   [WHERE search-condition | rowset-search-condition ] ■        (Embed)
   [SAMPLE sampling-method]
   [TRANSPOSE transpose-set [transpose-set]...
      [KEY BY key-colname]]...
   [SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
      [,colname [ASC[ENDING] | DESC[ENDING]]]...]
   [GROUP BY {colname | colnum} [,{colname | colnum}]...]
   [HAVING search-condition | rowset-search-condition ] ■       (Embed)
   [ORDER BY {colname | colnum} [ASC[ENDING] | DESC[ENDING]]
      [,{colname | colnum} [ASC[ENDING] | DESC[ENDING]]]...]
   [[FOR] access-option ACCESS]
   [IN {SHARE | EXCLUSIVE} MODE]
   [UNION [ALL] select-stmt]
```

```
select-list is:
   * | select-sublist [,select-sublist]...

select-sublist is:
   corr.* | [corr.] single-col [[AS]name] |
     [col-expr [[AS]name]

table-ref is:
     table [[AS] corr [(col-expr-list)]]

   | STREAM (table) [[AS] corr [(col-expr-list)]]
        [AFTER LAST ROW] ■

   | view  [[AS] corr [(col-expr-list)]]

   | STREAM (view) [[AS] corr [(col-expr-list)]]
        [AFTER LAST ROW] ■

   | (query-expr) [AS] corr [(col-expr-list)]

   | (delete-statement [RETURN select-list])
        [AS] corr [(col-expr-list)]
   | (update-statement [RETURN select-list])
        [AS] corr [(col-expr-list)] ■

   | joined-table


access-option is:
     READ UNCOMMITTED
   | READ COMMITTED
   | SERIALIZABLE
   | REPEATABLE READ
   | SKIP CONFLICT
   | STABLE

query-expr is:
   non-join-query-expr | joined-table

non-join-query-expr is:
   non-join-query-primary | query-expr UNION [ALL] query-term

query-term is:
   non-join-query-primary | joined-table

non-join-query-primary is:
   simple-table | (non-join-query-expr)

joined-table is:
   table-ref [NATURAL] [join-type] JOIN table-ref [join-spec]
  |table-ref CROSS JOIN table-ref

join-type is:
   INNER | LEFT [OUTER] | RIGHT [OUTER]
```

Pub/Sub

Pub/Sub

Pub/Sub

Embed

Embed

```
join-spec is:
   ON search-condition | rowset-search-condition ■

simple-table is:
     VALUES (row-value-const) [,(row-value-const)]...
   | TABLE table
   | SELECT [ALL | DISTINCT] select-list
        FROM table-ref [,table-ref]...
      | FROM ROWSET [rowset-size]
         (:array-name [,:array-name]...) ■
        [WHERE search-condition | rowset-search-condition]
        [SAMPLE sampling-method]
        [TRANSPOSE transpose-set [transpose-set]...
           [KEY BY key-colname]]...
        [SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
           [,colname [ASC[ENDING] | DESC[ENDING]]]...]
        [GROUP BY {colname | colnum} [,{colname |
colnum}]...]
        [HAVING search-condition | rowset-search-condition]]
        [[FOR] access-option ACCESS]
        [IN {SHARE | EXCLUSIVE} MODE]

row-value-const is:
     row-subquery
   | {expression | NULL} [,{expression | NULL}]...

sampling-method is:
     RANDOM percent-size
   | FIRST rows-size
              [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
                [,colname [ASC[ENDING] | DESC[ENDING]]]...]
   | PERIODIC rows-size EVERY number-rows ROWS
              [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
                [,colname [ASC[ENDING] | DESC[ENDING]]]...]

percent-size is:
     percent-result PERCENT [ROWS
       | {CLUSTERS OF number-blocks BLOCKS}]
   | BALANCE WHEN condition
       THEN percent-result PERCENT [ROWS]
       [WHEN condition THEN percent-result PERCENT [ROWS]]...
       [ELSE percent-result PERCENT [ROWS]] END

rows-size is:
     number-rows ROWS
   | BALANCE WHEN condition THEN number-rows ROWS
            [WHEN condition THEN number-rows ROWS]...
            [ELSE number-rows ROWS] END

transpose-set is:
   transpose-item-list AS transpose-col-list

transpose-item-list is:
   expression-list |(expression-list)[,(expression-list)]...
```

```
expression-list is:
   expression [,expression]...

transpose-col-list is:
   colname | (colname-list)

colname-list is:
   colname [,colname]..
```

Embed

`ROWSET FOR` *`size-and-index`*

Allowed only if you use rowsets in the SELECT statement. INPUT SIZE *`rowset-size-in`* and KEY BY *`row-id`* are allowed only if you specify *`rowset-search-condition`* in the where clause. OUTPUT SIZE *`rowset-size-out`* is allowed only with SELECT … INTO statements where *`variable-spec`* consists of rowset type host variables.

`INPUT SIZE` *`rowset-size-in`*

restricts the size of the input rowset to the specified size, which must be less than or equal to the allocated size for the rowset. The size is an integer literal (exact numeric literal), a dynamic parameter, or a host variable whose type is either unsigned short, signed short, unsigned long, or signed long in C and their corresponding equivalents in COBOL. If you do not specify the size, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section of the embedded SQL program.

`OUTPUT SIZE` *`rowset-size-out`*

restricts the size of the output rowset to the specified size which must be less than or equal to the allocated size for the rowset. The size is an integer literal (exact numeric literal) or a host variable whose type is signed long in C and its corresponding equivalent in COBOL. If you do not specify the size, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section of the embedded SQL program. This option is not supported in a cursor declaration. OUTPUT SIZE is supported only with SELECT ... INTO statements.

`KEY BY` *`row-id`*

is a zero-based index that identifies each row in the result set of a SELECT or FETCH statement with the particular *`search-condition`* in the WHERE clause that caused the row to be part of the result set. For example, if the *`row-id`* value for a certain row in the result set is 0 (zero), this row matches the *`search-condition`* in the first element of the host variable arrays (array index 0 in C, array index 1 in COBOL) in the WHERE clause.

For more information about rowsets, see the *SQL/MX Programming Manual for C and COBOL* ■

`[ANY N] | [FIRST N]`

specifies whether to select ANY of the first *N* rows or the `FIRST N` sorted rows. You must enclose `ANY N` or `FIRST N` in square brackets. `[FIRST N]` is different from `[ANY N]` only if you use ORDER BY on any of the columns in the select list to sort the result table of the SELECT statement. *N* is an unsigned numeric literal with no scale. If *N* is greater than the number of rows in the table, all rows are returned. `[ANY N]` and `[FIRST N]` are allowed in nested SELECT statements and on either side of a UNION operation.

`ALL | DISTINCT`

specifies whether to retrieve all rows whose columns are specified by the *select-list* (ALL) or only rows that are not duplicates (DISTINCT). Nulls are considered equal for the purpose of removing duplicates. The default is ALL.

select-list

specifies the columns or column expressions to select from the table references in the FROM clause.

`*`

specifies all columns in a table, view, joined table, or derived table determined by the evaluation of a query expression, as specified in the FROM clause.

corr`.*`

specifies all columns of specific table references by using the correlation name *corr* of the table references, as specified in the FROM clause. See Correlation Names on page 6-11.

corr`.`

specifies one column of specific table references by using the correlation name *corr* of the table reference, as specified in the FROM clause.

single-col `[[AS] name]`

specifies a column.

col-expr `[[AS]`*name*`]`

specifies a derived column determined by the evaluation of an SQL value expression in the list. By using the AS clause, you can associate a derived column *col-expr* with a *name*.

See the discussion of limitations in Considerations for Select List on page 2-350.

Embed

`INTO` *variable-spec* `[,`*variable-spec*`]...`

specifies host variables in which to return the values in the result row of the SELECT statement. The number of items in *select-list* must be equal to the number of specified host variables, and the data type of each source value must be compatible with the data type of its target host variable. The first value in the result row is assigned to the first host variable, the second value to the second variable, and so on.

C/COBOL You can specify rowset host variables in *`variable-spec`*. If you specify one rowset host variable, all specified host variables in the INTO list must be rowsets. Multiple rows can be returned when rowsets are used. If the number of rows returned is less than the length of the rowset, no error is displayed. However, if the number of rows returned exceeds the length of the rowset or the specified rowset-size-out, NonStop SQL/MX displays an error. ■

Use the INTO clause only for operations that are not union operations and return no more than one row. If the SELECT statement returns more than one row, use rowset host variables in the INTO list, or use a cursor.

C/COBOL `:`*`variable-name`*` [[INDICATOR] :`*`indicator-name`*`]`

is a variable specification—a host variable with, optionally, an indicator variable. A variable name begins with a colon (`:`). The values in the result row of the SELECT statement are returned in these host variables.

The data type of an indicator variable is exact numeric with a scale of 0. If the data returned in the host variable is null, the indicator parameter is set to a value less than zero. If character data returned is truncated, the indicator parameter is set to the length of the string in the database. ■

See single-row SELECT statements in the *SQL/MX Programming Manual for C and COBOL*. ■

`FROM `*`table-ref`*` [,`*`table-ref`*`]...`

specifies a list of tables, views, derived tables, or joined tables that determine the contents of an intermediate result table from which NonStop SQL/MX returns the columns you specify in *`select-list`*. To refer to a table or view, use one of these name types:

- Guardian physical name
- ANSI logical name
- DEFINE name

See Database Object Names on page 6-13.

If you specify only one *`table-ref`*, the intermediate result table consists of rows derived from that table reference. If you specify more than one *`table-ref`*, the intermediate result table is the cross-product of result tables derived from the individual table references.

```
     table [[AS] corr [(col-expr-list)]]
Pub/Sub   | STREAM (table) [[AS] corr [(col-expr-list)]]
        [AFTER LAST ROW] ■
   | view  [[AS] corr [(col-expr-list)]]
Pub/Sub   | STREAM (view) [[AS] corr [(col-expr-list)]]
        [AFTER LAST ROW] ■
   | (query-expr) [AS] corr [(col-expr-list)]
Pub/Sub   | (delete-statement [RETURN select-list])
        [AS] corr [(col-expr-list)]
   | (update-statement [RETURN select-list])
        [AS] corr [(col-expr-list)] ■
   | joined-table
```

specifies a *table-ref* as either a single table, view, derived table determined by the evaluation of a query expression, a joined table, a streaming table or view, or an embedded update or delete statement.

You can specify this optional clause for a table or view. This clause is required for a derived table:

[AS] *corr* [(*col-expr-list*)]

specifies a correlation name *corr* for the preceding table reference *table-ref* in the FROM clause. See Correlation Names on page 6-11.

col-expr [[AS]*name*] [,*col-expr* [[AS]*name*]]...

specifies the items in *col-expr-list*, a list of derived columns.

For the specification of a query expression, see the syntax diagram for *query-expr* on page 2-330.

Pub/Sub STREAM (*table*) [[AS] *corr* [(*col-expr-list*)]]

returns a continuous data stream from a table. A cursor opened on a continuous data stream never returns an end-of-data condition but blocks (waits) and resumes execution when new rows become available.

[[AS] *corr* [(*col-expr-list*)]]

specifies an optional correlation name *corr* and an optional column list for the preceding table reference in the FROM clause.

[AFTER LAST ROW]

causes the stream to skip all existing rows in the table and return only rows that are published after the stream's cursor is opened. ■

Pub/Sub STREAM (*view*) [[AS] *corr* [(*col-expr-list*)]]

returns a continuous data stream from a view.

[AFTER LAST ROW]

causes the stream to skip all existing rows in the view and return only rows that are published after the stream's cursor is opened. ■

Pub/Sub

(*delete-statement* [RETURN *select-list*])
[AS] *corr* [(*col-expr-list*)]

enables an application to read and delete rows with a single operation. For the syntax of *delete-statement*, see the DELETE Statement on page 2-162.

RETURN *select-list*

specifies the columns or column expressions returned from the deleted row. The items in the *select-list* can be either of these forms:

[OLD.]*

specifies the row from the OLD table exposed by the embedded DELETE. The OLD table refers to column values before the delete operation. NEW is not allowed.

An implicit `OLD.*` return list is assumed for an embedded delete operation that does not specify a RETURN list.

col-expr [[AS]*name*]

specifies a derived column determined by the evaluation of an SQL value expression in the list. Any column referred to in a value expression is from the row in the OLD table exposed by the embedded DELETE. The OLD table refers to column values before the delete operation.

By using the AS clause, you can associate a derived column *col-expr* with a name *name*.

[AS] *corr* [(*col-expr-list*)]

specifies an optional correlation name *corr* and an optional column list for the preceding items in the select list *RETURN select-list*. ■

Pub/Sub

(*update-statement* [RETURN *select-list*])
[AS] *corr* [(*col-expr-list*)]

enables an application to read and update rows with a single operation. For the syntax of *update-statement*, see the UPDATE Statement on page 2-385.

RETURN *select-list*

specifies the columns or column expressions returned from the updated row. The items in the *select-list* can be either of these forms:

`[OLD.| NEW.]*`

specifies the row from the OLD or NEW table exposed by the embedded UPDATE. The OLD table refers to column values before the update operation; the NEW table refers to column values after the update operation. If a column has not been updated, the NEW value is equivalent to the OLD value.

An implicit `NEW.*` return list is assumed for an embedded update operation that does not specify a RETURN list.

`col-expr` `[[AS]`*`name`*`]`

specifies a derived column determined by the evaluation of an SQL value expression in the list. Any column referred to in a value expression can be specified as from the row in the OLD table exposed by the embedded UPDATE or can be specified as being from the row in the NEW table exposed by the embedded UPDATE.

For example: `RETURN old.empno,old.salary,new.salary, (new.salary - old.salary)`.

By using the AS clause, you can associate a derived column *`col-expr`* with a name *`name`*.

`[AS]` *`corr`* `[(`*`col-expr-list`*`)]`

specifies an optional correlation name *`corr`* and an optional column list for the preceding items in the select list `RETURN` *`select-list`*.

For example:

```
RETURN old.empno,old.salary,new.salary,
   (new.salary - old.salary)
AS emp (empno, oldsalary, newsalary, increase). ■
```

`table-ref` `[NATURAL] [`*`join-type`*`] JOIN` *`table-ref`* `[`*`join-spec`*`]`

`join-type` is:
`CROSS |INNER | LEFT [OUTER] | RIGHT [OUTER]`

is a joined table. You specify the *`join-type`* by using the CROSS, INNER, OUTER, LEFT, and RIGHT keywords. If you omit the optional OUTER keyword and use LEFT or RIGHT in a join, NonStop SQL/MX assumes the join is an outer join.

If you specify a CROSS join as the *`join-type`*, you cannot specify a NATURAL join or a *`join-spec`*.

If you specify an INNER, LEFT, or RIGHT join as the *`join-type`* and you do not specify a NATURAL join, you must use an ON clause as the *`join-spec`*, as follows:

`ON` *`search-condition`*

specifies a *`search-condition`* for the join. Each column reference in *`search-condition`* must be a column that exists in either of the two result tables derived from the table references to the left and right of the JOIN keyword. A join of two rows in the result tables occurs if the condition is satisfied for those rows.

Embed

`ON` *`rowset-search-condition`*

specifies a *`rowset-search-condition`* for the join. The array of search conditions are evaluated successively, and for each condition a join of two rows in the result tables occurs if the condition is satisfied for those rows. Each column reference in *`rowset-search-condition`* must be a column that exists in either of the two result tables derived from the table references to the left and right of the JOIN keyword. ■

The type of join and the join specification if used determine which rows are joined from the two table references, as follows:

`table-ref` `CROSS JOIN` *`table-ref`*

joins each row of the left *`table-ref`* with each row of the right *`table-ref`*.

`table-ref` `NATURAL JOIN` *`table-ref`*

joins rows only where the values of all columns that have the same name in both tables match. This option is equivalent to NATURAL INNER.

`table-ref` `NATURAL LEFT JOIN` *`table-ref`*

joins rows where the values of all columns that have the same name in both tables match, plus rows from the left *`table-ref`* that do not meet this condition.

`table-ref` `NATURAL RIGHT JOIN` *`table-ref`*

joins rows where the values of all columns that have the same name in both tables match, plus rows from the right *`table-ref`* that do not meet this condition.

`table-ref` `JOIN` *`table-ref`* `ON`

joins only rows that satisfy the condition in the ON clause. This option is equivalent to INNER JOIN ... ON.

`table-ref` `LEFT JOIN` *`table-ref`* `ON`

joins rows that satisfy the condition in the ON clause, plus rows from the left *`table-ref`* that do not satisfy the condition.

table-ref RIGHT JOIN *table-ref* ON

joins rows that satisfy the condition in the ON clause, plus rows from the right *table-ref* that do not satisfy the condition.

The three ways a *simple-table* can be specified are:

```
  VALUES (row-value-const) [,(row-value-const)]...
| TABLE table
| SELECT [ALL | DISTINCT] select-list
     FROM table-ref [,table-ref]...
   | FROM ROWSET [rowset-size]
         (:array-name [,:array-name]...)
     [WHERE search-condition | rowset-search-condition]
     [SAMPLE sampling-method]
     [TRANSPOSE transpose-set [transpose-set]...
        [KEY BY key-colname]]...
     [SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
        [,colname [ASC[ENDING] | DESC[ENDING]]]...]
     [GROUP BY {colname | colnum} [,{colname | colnum}]...]
     [HAVING search-condition | rowset-search-condition]
     [[FOR] access-option ACCESS]
     [IN {SHARE | EXCLUSIVE} MODE]
```

A *simple-table* can be a table value constructor. It starts with the VALUES keyword followed by a sequence of row value constructors, each of which is enclosed in parentheses. A *row-value-const* is a list of expressions (or NULL) or a row subquery (a subquery that returns a single row of column values). An operand of an expression cannot reference a column (except when the operand is a scalar subquery returning a single column value in its result table).

The use of NULL as a *row-value-const* element is an SQL/MX extension.

A *simple-table* can be specified by using the TABLE keyword followed by a table name, which is equivalent to the query specification SELECT * FROM *table*.

A *simple-table* can be a query specification—that is, a SELECT statement consisting of SELECT ... FROM ... with optionally the WHERE, SAMPLE, TRANSPOSE, SEQUENCE BY, GROUP BY, and HAVING clauses. This form of a simple table is typically used in an INSERT, CREATE VIEW, or DECLARE CURSOR statement.

Embed FROM ROWSET *rowset-size*

restricts the size of the rowset-derived table to the specified size, which must be less than or equal to the allocated size for the rowset. The size, if specified, immediately follows the ROWSET keyword. The size is an unsigned integer or a host variable whose value is an unsigned integer. If you do not specify the size, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section.

`:array-name [,:array-name]...`

specifies a set of host variable arrays. Each *array-name* can be used like a column in the rowset-derived table. Each *array-name* can be any valid host language identifier with a data type that corresponds to an SQL data type. Precede each *array-name* with a colon (:) within an SQL statement.

For more information on rowsets and host variable arrays, see the *SQL/MX Programming Manual for C and COBOL*. ■

`WHERE search-condition`

specifies a `search-condition` for selecting rows. See Search Condition on page 6-108. The WHERE clause cannot contain an aggregate (set) function.

The `search-condition` is applied to each row of the result table derived from the table reference in the FROM clause or, in the case of multiple table references, the cross-product of result tables derived from the individual table references.

Each column you specify in `search-condition` is typically a column in this intermediate result table. In the case of nested subqueries used to provide comparison values, the column can also be an outer reference. See Subquery on page 6-112.

To comply with ANSI standards, NonStop SQL/MX does not move aggregate predicates from the WHERE clause to a HAVING clause and does not move non-aggregate predicates from the HAVING clause to the WHERE clause, as NonStop SQL/MP does.

Embed `WHERE rowset-search-condition`

specifies a `rowset-search-condition` for selecting rows. See Rowset Search Condition on page 6-110. The WHERE clause cannot contain an aggregate (set) function. The individual search conditions in `rowset-search-condition` are applied successively to the result table derived from the table reference in the FROM clause or, in the case of multiple table references, the cross-product of result tables derived from the individual table references. A row that matches any one of the individual search conditions is selected. If a row matches multiple search conditions, it is selected only once.

Each column you specify in rowset-search-condition is typically a column in this intermediate result table. In the case of nested subqueries used to provide comparison values, the column can also be an outer reference. See Subquery on page 6-112. ■

`SAMPLE` *`sampling-method`*

specifies the sampling method used to select a subset of the intermediate result table of a SELECT statement. Each of the methods uses a sampling size. The three sampling methods—random, first, and periodic—are specified as:

`RANDOM` *`percent-size`*

directs NonStop SQL/MX to choose rows randomly (each row having an unbiased probability of being chosen) without replacement from the result table. The sampling size is determined by using a percent of the result table.

`FIRST` *`rows-size`* `[SORT BY` *`colname`* `[,`*`colname`*`]...]`

directs NonStop SQL/MX to choose the first *`rows-size`* rows from the sorted result table. The sampling size is determined by using the specified number of rows.

```
PERIODIC rows-size EVERY number-rows ROWS
              [SORT BY colname [,colname]...]
```

directs NonStop SQL/MX to choose the first rows from each block (period) of contiguous sorted rows. The sampling size is determined by using the specified number of rows chosen from each block.

SAMPLE is an SQL/MX extension. See SET CATALOG Statement on page 2-366.

```
TRANSPOSE transpose-set [transpose-set]...
   [KEY BY key-colname]
```

specifies the *`transpose-sets`* and an optional key clause within a TRANSPOSE clause. You can use multiple TRANSPOSE clauses in a SELECT statement.

`transpose-item-list` `AS` *`transpose-col-list`*

specifies a *`transpose-set.`* You can use multiple transpose sets within a TRANSPOSE clause. The TRANSPOSE clause generates, for each row of the source table derived from the table reference or references in the FROM clause, a row for each item in each *`transpose-item-list`* of all the transpose sets.

The result table of a TRANSPOSE clause has all the columns of the source table plus a value column or columns, as specified in each *`transpose-col-list`* of all the transpose sets, and an optional key column *`key-colname`*.

`KEY BY` *`key-colname`*

optionally specifies an optional key column *`key-colname`*. It identifies which expression the value in the transpose column list corresponds to by its position in the *`transpose-item-list`*. *`key-colname`* is an SQL identifier. The data type is exact numeric, and the value is NOT NULL.

TRANSPOSE is an SQL/MX extension. See TRANSPOSE Clause on page 7-26.

```
SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
   [,colname [ASC[ENDING] | DESC[ENDING]]]...
```

specifies the order in which to sort the rows of the intermediate result table for calculating sequence functions. You must include a SEQUENCE BY clause if you include a sequence function in *select-list*. Otherwise, NonStop SQL/MX returns an error. Further, you cannot include a SEQUENCE BY clause if there is no sequence function in *select-list*.

colname

names a column in *select-list* or a column in a table reference in the FROM clause of the SELECT statement. *colname* is optionally qualified by a table, view, or correlation name; for example, CUSTOMER.CITY.

ASC | DESC

specifies the sort order. The default is ASC. When NonStop SQL/MX orders an intermediate result table on a column that can contain null, nulls are considered equal to one another but greater than all other non-null values.

```
GROUP BY [col-expr] {colname | colnum} [,{colname | colnum}]...]
```

specifies grouping columns *colname [,colname]...* that define a set of groups for the result table of the SELECT statement. These columns must appear in the list of columns in the table references in the FROM clause of the SELECT statement.

If you include a GROUP BY clause, the columns you refer to in the *select-list* must be either grouping columns or arguments of an aggregate (or set) function.

The grouping columns define a set of groups in which each group consists of rows with identical values in the specified columns. The column names can be qualified by a table or view name or a correlation name; for example, CUSTOMER.CITY.

For example, if you specify AGE, the result table contains one group of rows with AGE equal to 40 and one group of rows with AGE equal to 50. If you specify AGE and then JOB, the result table contains one group for each age and, within each age group, subgroups for each job code.

You can specify GROUP BY using ordinals to refer to the relative position within the SELECT list. For example, GROUP BY 3, 2, 1.

For grouping purposes, all nulls are considered equal to one another. The result table of a GROUP BY clause can have only one null group.

See Considerations for GROUP BY on page 2-351.

`HAVING` *search-condition*

specifies a *search-condition* to apply to each group of the grouped table resulting from the preceding GROUP BY clause in the SELECT statement. The GROUP BY clause, if one exists, must precede the HAVING clause in the SELECT statement.

To comply with ANSI standards, NonStop SQL/MX does not move aggregate predicates from the WHERE clause to a HAVING clause and does not move non-aggregate predicates from the HAVING clause to the WHERE clause, as NonStop SQL/MP does.

If there is no GROUP BY clause, the *search-condition* is applied to the entire table (which consists of one group) resulting from the WHERE clause (or the FROM clause if there is no WHERE clause).

In *search-condition*, you can specify any column as the argument of an aggregate (or set) function; for example, AVG (SALARY). An aggregate function is applied to each group in the grouped table.

A column that is not an argument of an aggregate function must be a grouping column. When you refer to a grouping column, you are referring to a single value because each row in the group contains the same value in the grouping column.

See Search Condition on page 6-108.

Embed `HAVING` *rowset-search-condition*

specifies a rowset search condition to apply to each group of the grouped table resulting from the preceding GROUP BY clause in the SELECT statement. The individual search conditions in the *rowset-search-condition* array are successively applied to each group. The GROUP BY clause, if one exists, must precede the HAVING clause in the SELECT statement. If there is no GROUP BY clause, *rowset-search-condition* is applied to the entire table (which consists of one group) resulting from the WHERE clause (or the FROM clause if there is no WHERE clause).

You can specify any column as the argument of an aggregate (or set) function; for example, AVG (SALARY). An aggregate function is applied to each group in the grouped table. A column that is not an argument of an aggregate function must be a grouping column. When you refer to a grouping column, you are referring to a single value because each row in the group contains the same value in the grouping column.

See Rowset Search Condition on page 6-110. ■

`[FOR]` *access-option* `ACCESS`

specifies the *access-option* when accessing data specified by the SELECT statement or by a table reference in the FROM clause derived from the evaluation of a query expression that is a SELECT statement. See Data Consistency and Access Options on page 1-8.

`READ UNCOMMITTED`

specifies that any data accessed need not be from committed rows. You can specify the SQL/MP extension BROWSE instead of READ UNCOMMITTED.

`READ COMMITTED`

specifies that any data accessed must be from committed rows.

`SERIALIZABLE | REPEATABLE READ`

specifies that the SELECT statement and any concurrent process (accessing the same data) execute as if the statement and the other process had run serially rather than concurrently.

`SKIP CONFLICT`

enables transactions to skip rows locked in a conflicting mode by another transaction. SKIP CONFLICT cannot be used in a SET TRANSACTION statement.

`STABLE`

specifies that the row being accessed is locked while it is processed, but concurrent use of the database is allowed.

The default access option is the isolation level of the containing transaction, which is determined according to the rules specified in Isolation Level on page 10-56.

`IN {SHARE | EXCLUSIVE} MODE`

specifies that either SHARE or EXCLUSIVE locks be used when accessing data specified by a SELECT statement or by a table reference in the FROM clause derived from the evaluation of a query expression that is a SELECT statement, and when accessing the index, if any, through which the table accesses occur.

`UNION [ALL]` *`select-stmt`*

specifies a set UNION operation between the result table of a SELECT statement and the result table of another SELECT statement.

The result of the UNION operation is a table that consists of rows belonging to either of the two contributing tables. If you specify UNION ALL, the table contains all the rows retrieved by each SELECT statement. Otherwise, duplicate rows are removed.

The select lists in the two SELECT statements of a UNION operation must have the same number of columns, and columns in corresponding positions within the lists must have compatible data types. The select lists must not be preceded by `[ANY` *`N`*`]` or `[FIRST` *`N`*`]`.

The number of columns in the result table of the UNION operation is the same as the number of columns in each select list. The column names in the result table of the UNION are the same as the corresponding names in the select list of the left SELECT statement. A column resulting from the UNION of expressions or constants has the name (EXPR).

See Considerations for UNION on page 2-351.

```
ORDER BY {colname | colnum} [ASC[ENDING] | DESC[ENDING]]
   [,{colname | colnum} [ASC[ENDING] | DESC[ENDING]]]...
```

specifies the order in which to sort the rows of the final result table.

colname

names a column in *select-list* or a column in a table reference in the FROM clause of the SELECT statement. *colname* is optionally qualified by a table, view, or correlation name; for example, CUSTOMER.CITY. If a column has been aliased to another name you must use the alias name.

colnum

specifies a column by its position in *select-list*. Use *colnum* to refer to unnamed columns, such as derived columns.

ASC | DESC

specifies the sort order. The default is ASC. For ordering a result table on a column that can contain null, nulls are considered equal to one another but greater than all other non-null values.

See Considerations for ORDER BY on page 2-351.

Considerations for SELECT

Embed

Multiple Row and Single Row SELECT Statements

Use a multiple row SELECT statement (without the INTO clause) within a CREATE VIEW statement to specify views. You can also use it to query a database within MXCI. In embedded SQL, a multiple row SELECT statement can also be a cursor specification—a special case of a query expression. For more information, see the *SQL/MX Programming Manual for C and COBOL*. See syntax for `query-expr` on page 2-331.

In embedded SQL, if rowset host variables are not used, you use a single row SELECT statement (with the INTO clause) to retrieve only one row. If the SELECT statement (with the INTO clause) retrieves more than one row, and rowsets are not used, NonStop SQL/MX raises an error. If rowset host variables are used in the INTO clause multiple rows can be retrieved with a multiple row SELECT statement. ■

Authorization Requirements

SELECT requires authority to read all views and tables referred to in the statement, including the underlying tables of views referred to in the statement.

Transactions

Queries on audited tables must be performed within a transaction unless the SELECT statement uses READ UNCOMMITTED access.

Locking Modes

When specifying the locking mode for a SELECT statement:

- Use SHARE mode when the process reads data but does not modify it. Specifying READ COMMITTED access and SHARE mode ensures a higher level of concurrency.
- Use EXCLUSIVE mode when the process reads data and then modifies it with DELETE or UPDATE. Requesting EXCLUSIVE locks on the SELECT prevents other processes from acquiring SHARE locks on the accessed rows between the time of the SELECT and the time of the subsequent DELETE or UPDATE. Such locks by other processes would prevent the process from escalating its own SHARE locks to the EXCLUSIVE locks required for a DELETE or UPDATE operation, causing the process to wait or timeout.
- Do not specify the IN clause for READ UNCOMMITTED access. If you omit the IN clause for other access options, SQL uses SHARE until an attempt is made to modify the data, and then escalates the lock to EXCLUSIVE.

Locking modes are relevant only to SELECT operations that use a cursor. In a standalone SELECT statement, locks are maintained only for the duration of the SELECT.

Use of Views With SELECT

When a view is referenced in a SELECT statement, the specification that defines the view is combined with the statement. The combination can cause the SELECT statement to be invalid. If you receive an error message that indicates a problem but the SELECT statement seems to be valid, check the view definition.

For example, suppose that the view named AVESAL includes column A defined as AVG (X). The SELECT statement that contains MAX (A) in its select list is invalid because the select list actually contains MAX (AVG (X)), and an aggregate function cannot have an argument that includes another aggregate function.

Join Limits

Note. HP recommends that you limit the number of tables in a join to a maximum of 64, which includes base tables of views referenced in joins. Queries with joins that involve a larger number of tables are not guaranteed to compile.

Object Names in SELECT

You can use fully qualified Guardian names only in the FROM clause of a SELECT statement.

AS and ORDER BY Conflicts

When you use the AS verb to rename a column in a SELECT statement, and the ORDER BY clause uses the original column name, the query will fail. If a column has been aliased to another name you must use the alias name. This type of query is not supported by the ANSI standard.

Pub/Sub

Stream Access Restrictions

- SELECT statements can access only one table with stream access except for unions that allow both SELECT statements to use stream access. However, you must use UNION ALL when using stream access with unions.
- Streams assume parallel access to data; that is, if a table is partitioned and you attempt to access it as a stream, parallel access to partitions is required. If you try to access a stream when the default ATTEMPT_ASYNCHRONOUS_ACCESS is set to OFF, NonStop SQL/MX returns an error. See ATTEMPT_ASYNCHRONOUS_ ACCESS on page 10-66
- You cannot join two streams.
- Aggregate functions are not supported on streams, and therefore no GROUP BY or HAVING clauses are valid on streams.
- Sort operations are not supported on streams. Therefore, you cannot use DISTINCT, UNION DISTINCT, or ORDER BY unless supported by an index. You can use a secondary index for accessing an ordered stream only if the columns in the index definition include all the columns of the base table accessed as a stream that are referenced in its WHERE clause.
- A query expression that serves as a data source for an INSERT statement cannot specify stream access.
- A delete or update statement that is not embedded as a table reference cannot specify stream access. For example, the statement `DELETE FROM STREAM(tab1)` is not valid.
- If your application must handle a fast rate of publishing into the stream, or publishes rows in very large transactions, it must be able to handle stream

overflows. See the run-time limits on streams in the *SQL/MX Queuing and Publish/Subscribe Services* for details.

- You cannot use streams with nonaudited tables.
- Stream access within compound statements is not supported. ■

Joining the Results of an Embedded Delete or Update

Pub/Sub

SQL/MX Release 2.x enables you to join another table with the results of an embedded delete or embedded update. For more information, see *SQL/MX Queuing and Publish/Subscribe Services*. ■

Restrictions on Embedded Deletes and Updates

Pub/Sub

These restrictions apply to embedded deletes and updates:

- You cannot use an embedded delete or update with a union statement—not even UNION ALL.
- When you use an embedded delete or update with a join, the join predicate must use the other table's primary key.
- You cannot join an embedded delete or update with a stream.
- You cannot join an embedded delete or update with another embedded delete or update.
- The table referenced by an embedded delete or update cannot be referenced again in the same statement.
- Rowsets cannot be used as the selection predicate for an embedded delete or update.
- An embedded deletes or update cannot be used with a compound statement.
- You cannot use an embedded delete or update with aggregates (for example, GROUP BY, HAVING, or DISTINCT).
- You cannot sort an embedded delete or update. Therefore, you cannot use DISTINCT or ORDER BY unless they are supported by an index. You can use a secondary index to access an ordered embedded delete or update only if the columns in the index definition include all the columns of the base table accessed as an embedded delete or update that are referenced in the WHERE clause. ■

DISTINCT Aggregate Functions

An aggregate function can accept an argument specified as DISTINCT, which eliminates duplicate values before the aggregate function is applied. Only one DISTINCT aggregate function is allowed at each level of a SELECT statement. Multiple DISTINCT aggregates are allowed if they are on the same column, but are not permitted on different columns. Exceptions to this rule are SQL/MX extensions for which DISTINCT is unnecessary and include:

- MIN and MAX functions
- Aggregate functions with unique columns or expressions, such as primary keys or UNIQUE constraints

These aggregate functions do not contribute to the count of DISTINCT aggregate functions in the query, thus permitting you to specify them more than once or in addition to another DISTINCT aggregate function in a query.

Considerations for Select List

C/COBOL

- If a column in a select list has datetime or interval data type, you must use the CAST function to convert the column to a character string in an embedded SQL program. You must also specify the length of the target host variable (or the length – 1 in the case of a C program) as part of the CAST conversion. ■
- The `*` and *`corr.*`* forms of a *`select-list`* specification are convenient for use in MXCI. However, such specifications make the order of columns in the SELECT result table dependent on the order of columns in the current definition of the referenced tables or views.
- A *`col-expr`* is a single column name or a derived column. A derived column is an SQL value expression; its operands can be numeric, string, datetime, or interval literals, columns, functions (including aggregate functions) defined on columns, scalar subqueries, CASE expressions, or CAST expressions. Any single columns named in *`col-expr`* must be from tables or views specified in the FROM clause. For a list of aggregate functions, see Aggregate (Set) Functions on page 8-1.
- If *`col-expr`* is a single column name, that column of the SELECT result table is a named column. All other columns are unnamed columns in the result table (and have the (EXPR) heading) unless you use the AS clause to specify a name for a derived column.
- You can specify SYSKEY as an item in the *`select-list`*. A SYSKEY is a primary key defined by NonStop SQL/MX rather than by the user; it is the first column in a table, and its data type depends on the organization of the table's underlying file: key-sequenced or entry-sequenced. (NonStop SQL/MX supports only key-sequenced tables.) If you want to select the SYSKEY column from more than one result table, you must qualify SYSKEY; for example, EMPLOYEE.SYSKEY.

Considerations for SEQUENCE BY

If you include both SEQUENCE BY and GROUP BY clauses in the same SELECT statement, the values of the sequence functions must be computed first and then become input for the aggregate functions in the statement. For more information, see SEQUENCE BY Clause on page 7-19.

Considerations for GROUP BY

- If you include a GROUP BY clause, the columns you refer to in the *select-list* must be either grouping columns or arguments of an aggregate (or set) function. For example, if AGE is not a grouping column, you can refer to AGE only as the argument of a function, such as AVG (AGE).
- If you do not include a GROUP BY clause but you specify an aggregate function in the *select-list*, all rows of the result table form the one and only group. The result of AVG, for example, is a single value for the entire table.
- The GROUP BY clause must precede a HAVING clause.
- If the value of `col-expr` is a numeric constant, it refers to the position of the select list item and is treated as the current GROUP BY using the ordinal feature.
- You can specify GROUP BY using ordinals to refer to the relative position within the SELECT list. For example, GROUP BY 3, 2, 1.

Considerations for ORDER BY

When you specify an ORDER BY clause and its ordering columns, consider this:

- If you specify DISTINCT, the ordering column must be in *select-list*.
- If you specify a GROUP BY clause, the ordering column must also be a grouping column.
- If an ORDER BY clause applies to a union of SELECT statements, the ordering column must be explicitly referenced, and not within an aggregate function or an expression, in the *select-list* of the leftmost SELECT statement.
- SQL does not guarantee a specific or consistent order of rows unless you specify an ORDER BY clause. ORDER BY can reduce performance, however, so use it only if you require a specific order.

Considerations for UNION

Suppose that the contributing SELECT statements are named SELECT1 and SELECT2, the contributing tables resulting from the SELECT statements are named TABLE1 and TABLE2, and the table resulting from the UNION operation is named RESULT.

Characteristics of the UNION Columns

For columns in TABLE1 and TABLE2 that contribute to the RESULT table:

- If both columns contain character strings, the corresponding column in RESULT contains a character string whose length is equal to the greater of the two contributing columns.

- If both columns contain variable-length character strings, RESULT contains a variable-length character string whose length is equal to the greater of the two contributing columns.
- If any one of the operands is a character string constant, the corresponding column in the RESULT table contains a variable-length character string whose length is equal to the greater of the two contributing operands/columns.
- If both columns are of exact numeric data types, RESULT contains an exact numeric value whose precision and scale are equal to the greater of the two contributing columns.
- If both columns are of approximate numeric data types, RESULT contains an approximate numeric value whose precision is equal to the greater of the two contributing columns.
- If both columns are of datetime data type (DATE, TIME, or TIMESTAMP), the corresponding column in RESULT has the same data type.
- If both columns are INTERVAL data type and both columns are either year-month or day-time, RESULT contains an INTERVAL value whose range of fields is the most significant start field to the least significant end field of the INTERVAL fields in the contributing columns. (The year-month fields are YEAR and MONTH. The day-time fields are DAY, HOUR, MINUTE, and SECOND.)

  For example, suppose that the column in TABLE1 has the data type INTERVAL HOUR TO MINUTE, and the column in TABLE2 has the data type INTERVAL DAY TO HOUR. The data type of the column resulting from the union operation is INTERVAL DAY TO MINUTE.
- If both columns are described with NOT NULL, the corresponding column of RESULT cannot be null. Otherwise, the column can be null.

ORDER BY Clause and the UNION Operator

In a query containing a UNION operator, the ORDER BY clause defines an ordering on the result of the UNION. In this case, the SELECT statement cannot have an individual ORDER BY clause.

You can specify an ORDER BY clause only as the last clause following the final SELECT statement (SELECT2 in this example). The ORDER BY clause in RESULT specifies the ordinal position of the sort column either by using an integer or by using the column name from the select list of SELECT1.

This SELECT statement shows correct use of the ORDER BY clause:

```
SELECT A FROM T1 UNION SELECT B FROM T2 ORDER BY A
```

This SELECT statement is incorrect because the ORDER BY clause does not follow the final SELECT statement:

```
SELECT A FROM T1 ORDER BY A UNION SELECT B FROM T2
```

This SELECT statement is also incorrect:

```
SELECT A FROM T1 UNION (SELECT B FROM T2 ORDER BY A)
```

Because the subquery (SELECT B FROM T2...) is processed first, the ORDER BY clause does not follow the final SELECT.

GROUP BY Clause, HAVING Clause, and the UNION Operator

In a query containing a UNION operator, the GROUP BY or HAVING clause is associated with the SELECT statement it is a part of (unlike the ORDER BY clause, which can be associated with the result of a UNION operation). The groups are visible in the result table of the particular SELECT statement. The GROUP BY and HAVING clauses cannot be used to form groups in the result of a UNION operation.

UNION ALL and Associativity

The UNION ALL operation is left associative, meaning that these two queries return the same result:

```
(SELECT * FROM TABLE1 UNION ALL
  SELECT * FROM TABLE2) UNION ALL SELECT * FROM TABLE3;
```

```
SELECT * FROM TABLE1 UNION ALL
   (SELECT * FROM TABLE2 UNION ALL SELECT * FROM TABLE3);
```

If both the UNION ALL and UNION operators are present in the query, the order of evaluation is always from left to right. A parenthesized union of SELECT statements is evaluated first, from left to right, followed by the remaining union of SELECT statements.

Access Modes and the UNION Operator

In a query containing the UNION operator, if you specify an access option for the second operand before the ORDER BY clause (or if the UNION has no ORDER BY clause) and you do not specify an option for the first operand, the first operand inherits the session's transaction isolation level setting. If this setting is different from the one you specified for the second operand, NonStop SQL/MX issues a warning. For example:

```
SELECT common.isma_no FROM sdcommon common
 WHERE common.sec_status='L'
UNION
SELECT main.isma_no FROM sdmain main
 WHERE main.iss_eligible='Y'
 FOR READ UNCOMMITTED ACCESS
ORDER BY 1 ASCENDING;
```

This statement will receive a warning:

```
*** WARNING[3192] Union operands sdcommon common and sdmain
main have different transaction access/lock modes.
```

If you want the access you specified for the second operand to apply to both SELECT items in this type of query, use one of these strategies:

- Specify the desired access mode for each SELECT:

  ```
  SELECT common.isma_no FROM sdcommon common
   WHERE common.sec_status='L'
   FOR READ UNCOMMITTED ACCESS
  UNION
  SELECT main.isma_no FROM sdmain main
   WHERE main.iss_eligible='Y'
   FOR READ UNCOMMITTED ACCESS
  ORDER BY 1 ASCENDING;
  ```

- Use a table subquery to enclose the UNION, and apply the access mode to the main query. This statement receives a warning because NonStop SQL/MX treats the access mode on the second SELECT as applicable only to that second SELECT:

  ```
  SELECT a
    from t046a where b=1
    UNION
    SELECT b from t046b where a=2
    for browse access;
  ```

  This statement uses a table subquery to apply the access mode to both queries:

  ```
  SELECT c from
  (SELECT a from t046a where b=1
    UNION
    SELECT b from t046b where a=2) as t(c)
    for browse access;
  ```

- Specify the access mode after the ORDER BY clause:

```
SELECT common.isma_no
  from sdcommon common
  where common.sec_status='L'
UNION
SELECT main.isma_no
  from sdmain main
  where main.iss_eligible='Y'
ORDER BY 1 ascending for browse access;
```

For more about the effect of UNION on SELECT statements, including its effect on performance, see the *SQL/MX Query Guide*.

MXCI Examples of SELECT

- Retrieve information from the EMPLOYEE table for employees with a job code greater than 500 and who are in departments with numbers less than or equal to 3000, displaying the results in ascending order by job code:

```
SELECT jobcode, deptnum, first_name, last_name, salary
FROM persnl.employee
WHERE jobcode > 500 AND deptnum <= 3000
ORDER BY jobcode
READ UNCOMMITTED ACCESS;

JOBCODE  DEPTNUM  FIRST_NAME       LAST_NAME        SALARY
-------  -------  ---------------  -----------  ----------
    600  1500     JONATHAN         MITCHELL       32000.00
    600  1500     JIMMY            SCHNEIDER      26000.00
    900  2500     MIRIAM           KING           18000.00
    900  1000     SUE              CRAMER         19000.00
  . . .
```

In this example, because of READ UNCOMMITTED access, the query does not wait for other concurrent processes to commit rows.

- Display selected rows grouped by job code in ascending order:

```
SELECT jobcode, AVG(salary)
FROM persnl.employee
WHERE jobcode > 500 AND deptnum <= 3000
GROUP BY jobcode
ORDER BY jobcode;

JOBCODE  EXPR
-------  ----------------------
    600                29000.00
    900                25100.00

--- 2 row(s) selected.
```

This select list contains only grouping columns and aggregate functions. Each row of the output summarizes the selected data within one group.

- Select data from more than one table by specifying the table names in the FROM clause and specifying the condition for selecting rows of the result in the WHERE clause:

```
SELECT jobdesc, first_name, last_name, salary
FROM persnl.employee E, persnl.job J
WHERE E.jobcode = J.jobcode AND
  E.jobcode IN (900, 300, 420);

JOBDESC       FIRST_NAME    LAST_NAME        SALARY
------------  ------------  ---------------  -----------
SALESREP      TIM           WALKER              32000.00
SALESREP      HERBERT       KARAJAN             29000.00
...
ENGINEER      MARK          FOLEY               33000.00
ENGINEER      MARIA         JOSEF               18000.10
...
SECRETARY     BILL          WINN                32000.00
SECRETARY     DINAH         CLARK               37000.00
...

--- 27 row(s) selected.
```

This type of condition is sometimes referred to as a join predicate. The query first joins the EMPLOYEE and JOB tables by combining each row of the EMPLOYEE table with each row of the JOB table; the intermediate result is the Cartesian product of the two tables.

This join predicate specifies that any row (in the intermediate result) with equal job codes is included in the result table. The WHERE condition further specifies that the job code must be 900, 300, or 420. All other rows are eliminated.

The four logical steps that determine the intermediate and final results of the previous query are:

1. Join the tables.

EMPLOYEE Table			**JOB Table**	
EMPNUM ...	JOBCODE ...	SALARY	JOBCODE	JOBDESC

2. Drop rows with unequal job codes.

EMPLOYEE Table			**JOB Table**	
EMPNUM ...	JOBCODE ...	SALARY	JOBCODE	JOBDESC
1	100	175500	100	MANAGER
...	...	...	...	...
75	300	32000	300	SALESREP
...	...	...	...	...
178	900	28000	900	SECRETARY
...	...	...	...	...

EMPLOYEE Table			JOB Table	
207	420	33000	420	ENGINEER
. . .	. . .	. . .	. . .	. . .
568	300	39500	300	SALESREP

3. Drop rows with job codes not equal to 900, 300, or 420.

EMPLOYEE Table			JOB Table	
EMPNUM ...	JOBCODE ...	SALARY	JOBCODE	JOBDESC
75	300	32000	300	SALESREP
. . .	. . .	. . .	. . .	. . .
178	900	28000	900	SECRETARY
. . .	. . .	. . .	. . .	. . .
207	420	33000	420	ENGINEER
. . .	. . .	. . .	. . .	. . .
568	300	39500	300	SALESREP

4. Process the select list, leaving only four columns.

JOBDESC	FIRST_NAME	LAST_NAME	SALARY
SALESREP	TIM	WALKER	32000
. . .	. . .	. . .	. . .
SECRETARY	JOHN	CHOU	28000
. . .	. . .	. . .	. . .
ENGINEER	MARK	FOLEY	33000
. . .	. . .	. . .	. . .
SALESREP	JESSICA	CRINER	39500

The final result is shown in the output:

```
JOBDESC       FIRST_NAME    LAST_NAME         SALARY
------------  ------------  ---------------   -----------
SALESREP      TIM           WALKER               32000.00
...
SECRETARY     JOHN          CHOU                 28000.00
...
```

- Select from three tables, group the rows by job code and (within job code) by department number, and order the groups by the maximum salary of each group:

```
SELECT E.jobcode, E.deptnum, MIN (salary), MAX (salary)
FROM persnl.employee E,
  persnl.dept D, persnl.job J
WHERE E.deptnum = D.deptnum AND E.jobcode = J.jobcode
  AND E.jobcode IN (900, 300, 420)
GROUP BY E.jobcode, E.deptnum
ORDER BY 4;
```

```
JOBCODE  DEPTNUM  (EXPR)       (EXPR)
-------  -------  -----------  -----------
    900     1500     17000.00     17000.00
    900     2500     18000.00     18000.00
    ...
    300     3000     19000.00     32000.00
    900     2000     32000.00     32000.00
    ...
    300     3200     22000.00     33000.10
    420     4000     18000.10     36000.00
    ...

--- 16 row(s) selected.
```

Only job codes 300, 420, and 900 are selected. The minimum and maximum salary for the same job in each department are computed, and the rows are ordered by maximum salary.

- Select from two tables that have been joined by using an INNER JOIN on matching part numbers:

```
SELECT OD.*, P.*
FROM sales.odetail OD INNER JOIN sales.parts P
ON OD.partnum = P.partnum;
```

```
Order/Num   Part/Num  Unit/Price    Qty/Ord     Part/Num
Part Description    PRICE         Qty/Avail
----------  --------  ------------  ----------  --------
------------------  ------------  -----------
    400410       212      2450.00          12       212
PCSILVER, 20 MB         2500.00        3525

    500450       212      2500.00           8       212
PCSILVER, 20 MB         2500.00        3525

    100210       244      3500.00           3       244
PCGOLD, 30 MB           3000.00        4426

    800660       244      3000.00           6       244
PCGOLD, 30 MB           3000.00        4426
       ...       ...          ...         ...       ...
...                          ...          ...

--- 72 row(s) selected.
```

- Select from three tables and display them in employee number order. Two tables are joined by using a LEFT JOIN on matching department numbers, then an additional table is joined on matching jobcodes:

```
SELECT empnum, first_name, last_name, deptname, location,
jobdesc
  FROM employee e LEFT JOIN dept d ON e.deptnum = d.deptnum
    LEFT JOIN job j ON e.jobcode = j.jobcode
  ORDER BY empnum;
```

- Suppose that the JOB_CORPORATE table has been created from the JOB table by using the CREATE LIKE statement. Form the union of these two tables:

```
SELECT * FROM job UNION SELECT * FROM job_corporate;

JOBCODE  JOBDESC
-------  ------------------
    100  MANAGER
    200  PRODUCTION SUPV
    250  ASSEMBLER
    300  SALESREP
    400  SYSTEM ANALYST
    420  ENGINEER
    450  PROGRAMMER
    500  ACCOUNTANT
    600  ADMINISTRATOR
    900  SECRETARY
    100  CORP MANAGER
    300  CORP SALESREP
    400  CORP SYSTEM ANALYS
    500  CORP ACCOUNTANT
    600  CORP ADMINISTRATOR
    900  CORP SECRETARY

--- 16 row(s) selected.
```

- Present two ways to select the same data submitted by customers from California.

The first way:

```
SELECT OD.ordernum, SUM (qty_ordered * price)
FROM sales.parts P, sales.odetail OD
WHERE OD.partnum = P.partnum AND OD.ordernum IN
  (SELECT O.ordernum
   FROM sales.orders O, sales.customer C
   WHERE O.custnum = C.custnum AND state = 'CALIFORNIA')
GROUP BY OD.ordernum;

ORDERNUM    (EXPR)
----------  ---------------------
    200490                1030.00
    300350               71025.00
    300380               28560.00
--- 3 row(s) selected.
```

The second way:

```
SELECT OD.ordernum, SUM (qty_ordered * price)
FROM sales.parts P, sales.odetail OD
WHERE OD.partnum = P.partnum AND OD.ordernum IN
  (SELECT O.ordernum
   FROM sales.orders O
   WHERE custnum IN
    (SELECT custnum
     FROM sales.customer
     WHERE state = 'CALIFORNIA'))
```

```
GROUP BY OD.ordernum;

ORDERNUM    (EXPR)
----------  ---------------------
    200490                1030.00
    300350               71025.00
    300380               28560.00
--- 3 row(s) selected.
```

The price for the total quantity ordered is computed for each order number.

- Show employees, their salaries, and the percentage of the total payroll that their salaries represent. Note the subquery as part of the expression in the select list:

```
SELECT empnum, first_name, last_name, salary,
CAST(salary * 100 / (SELECT SUM(salary) FROM persnl.employee)
  AS NUMERIC(4,2))
FROM persnl.employee
ORDER BY salary, empnum;

Employee/Number  First Name       Last Name
salary       (EXPR)
---------------  ---------------  --------------------
-----------  -------
            209  SUSAN            CHAPMAN
17000.00       .61
            235  MIRIAM           KING
18000.00       .65
            224  MARIA            JOSEF
18000.10       .65
...
             23  JERRY            HOWARD
137000.10     4.94
             32  THOMAS           RUDLOFF
138000.40     4.98
              1  ROGER            GREEN
175500.00     6.33
...

--- 62 row(s) selected.
```

C Examples of SELECT

- Use a single-row SELECT statement:

```
EXEC SQL
  SELECT LAST_NAME, FIRST_NAME INTO :hv_lname, :hv_fname
  FROM EMPLOYEE WHERE EMPNUM = 1234;
```

- Use an indicator variable:

```
EXEC SQL BEGIN DECLARE SECTION;
...
short ihv_salary;              /* Indicator variable */
float hv_salary;
EXEC SQL END DECLARE SECTION;
```

```
...
EXEC SQL
  SELECT SALARY INTO :hv_salary INDICATOR :ihv_salary
  FROM EMPLOYEE WHERE EMPNUM = 1234;
...
```

COBOL Examples of SELECT

- Use a single-row SELECT statement:

```
EXEC SQL
  SELECT LAST_NAME, FIRST_NAME INTO :hv-lname, :hv-fname
  FROM EMPLOYEE WHERE EMPNUM = 1234
END-EXEC.
```

- This example uses an indicator variable:

```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   ...
01 ihv-salary   PIC S9(4) comp.
01 hv-salary    PIC 9(6)V9(2) comp.
   EXEC SQL END DECLARE SECTION END-EXEC.
   ...
   EXEC SQL
     SELECT SALARY INTO :hv-salary INDICATOR :ihv-salary
     FROM EMPLOYEE WHERE EMPNUM = 1234
   END-EXEC.
   ...
```

Publish/Subscribe Examples of SELECT

Suppose that these SQL/MP tables and index (and the metadata mappings) have been created:

```
CREATE TABLE $db.dbtab.tab1 (a INT, b INT, c INT);
CREATE TABLE $db.dbtab.tab2 (a INT, b INT, c INT);
CREATE INDEX $db.dbtab.itab1 ON tab1(b, c);

CREATE SQLMP ALIAS cat.sch.tab1 $db.dbtab.tab1;
CREATE SQLMP ALIAS cat.sch.tab2 $db.dbtab.tab2;
```

- These examples show stream access, ordering the access by using an ORDER BY clause, and selecting entries by using a WHERE clause:

```
SET NAMETYPE ANSI;
SET SCHEMA cat.sch;

SELECT * FROM STREAM(tab1);
SELECT * FROM STREAM(tab1)
WHERE b = 1;

SELECT b, c FROM STREAM(tab1)
ORDER BY b;

SELECT b, c FROM STREAM(tab1)
WHERE c > 1 ORDER BY b;
```

For more information, see ordered streams in *SQL/MX Queuing and Publish/Subscribe Services*.

- These examples show join operations with a base table and a stream:

```
SELECT *
FROM tab1, (SELECT * FROM STREAM(tab2)) AS tab2
WHERE tab1.a = tab2.a;
SELECT * FROM STREAM(tab1), tab2
WHERE tab1.a = tab2.a;
```

The two preceding queries yield identical results.

- This example shows union operations with streams:

```
SELECT * FROM STREAM(tab1)
UNION ALL
SELECT * FROM STREAM(tab2);
```

- These examples show the embedded delete statement as a table reference:

```
SELECT * FROM (DELETE FROM tab1) AS tab1;
SELECT * FROM (DELETE FROM tab1) AS tab1
ORDER BY b;
SELECT * FROM (DELETE FROM tab1) AS tab1
WHERE b = 1;
SELECT * FROM (DELETE FROM tab1) AS tab1
WHERE c > 1 ORDER BY b;
```

- This example shows a return list in an embedded delete:

```
SELECT * FROM (DELETE FROM tab1 WHERE a > 1
               RETURN OLD.*) AS tab1;
```

- This example shows a return list in an embedded update:

```
SELECT * FROM
  (UPDATE tab1 SET a = a + 1
   WHERE a > 1 RETURN OLD.a, NEW.a)
   AS tab1(old_a, new_a);
```

- This example shows the SKIP CONFLICT access:

```
SELECT * FROM tab1 FOR SKIP CONFLICT ACCESS;

SELECT * FROM tab1, tab2
WHERE tab1.a = tab2.a
FOR SKIP CONFLICT ACCESS;
```

SELECT ROW COUNT Statement

The SELECT ROW COUNT statement is used to retrieve the count of rows from an SQL/MX table.

The `SELECT ROW COUNT ()` query is sent to each disk process where a partition of the table resides and the count of rows is computed as a parallel SQL operation. Because the operation uses a stored count maintained for each partition of a table, the SELECT ROW COUNT statement is an efficient way to obtain a row count.

```
SELECT ROW COUNT FROM table
```

`table`

specifies the name of the table for which you want to determine the number of rows.

Considerations for SELECT ROW COUNT

- The SELECT ROW COUNT statement can be issued from any interface where an existing SELECT statement is allowed.
- Other clauses, such as, WHERE, ORDER by, and GROUP by, are not allowed with this SELECT statement.
- *`table`* must be an SQL/MX base table.
- SQL/MP tables are not supported.
- When you use the EXPLAIN statement with a SELECT ROW COUNT statement, the result displays the count operator as `disk_label_stats`.
- The count is returned as if the target table is accessed in READ UNCOMMITTED mode. This means that an inaccurate row count could be returned because of transactions that have not yet committed.

Limitations of SELECT ROW COUNT

The SELECT ROW COUNT statement fails with an error 8022 (Stored row count is invalid) under any of the following conditions:

- The target table was created with an SQL version earlier than R3.0.
- The target table was created with an SQL DUP operation.
- The target table was created with a BACKUP RESTORE operation.

- Records have been added, deleted, or moved from one or more partitions of the target table by a MODIFY operation (in this case, only the affected partitions are marked with an invalid row count).

If partitions of a table are marked with an invalid row count, the row count can be reset by deleting all the rows from the affected partition(s) using either a DELETE or PURGEDATA statement.

Note. An invalid row count on one or more partitions only affects the ability of the SELECT ROW COUNT statement to return a valid count. An invalid row count does not affect any other SQL/MX operations and can be safely ignored. In such cases, a SELECT statement with COUNT(*) can be used instead of returning a row count.

Example of SELECT ROW COUNT

- This example selects the row count from the `persnl.employee` table:

```
SELECT ROW COUNT FROM persnl.employee;

(EXPR)

---------------------------------------
                               11487
```

SET Statement

Considerations for SET Statement

The SET statement is used with BEFORE triggers to assign values to variables representing columns in the SQL/MX table to be modified by the triggering action.

```
SET correlation-name.column-name = value-expression;
```

correlation-name

is the name of the new row that correlates to the row to be modified.

column-name

is the name of the new column that correlates to the column to be modified.

value-expression

is any valid SQL expression.

Considerations for SET Statement

The SET statement can appear only as an action of a BEFORE trigger. The left side of the assignment can specify only a column in the correlation name of the new row. The right side of the assignment can be any valid SQL expression (in particular, it can include subqueries).

In a BEFORE-type trigger action, any column can be updated by SET, including primary and clustering key columns.

SET CATALOG Statement

The SET CATALOG statement sets the default logical catalog for unqualified schema names for the current SQL session.

Embed The SET CATALOG statement sets the default catalog for unqualified schema names in all dynamic statements within the control flow scope of an embedded SQL program for the current SQL session. ■

```
SET CATALOG default-catalog-name
```

default-catalog-name

specifies the name of the catalog. See Catalogs on page 6-3.

MXCI *default-catalog-name* is an SQL identifier. For example, you can use `MYCATALOG` or `mycatalog` or a delimited identifier `"my catalog"`. See Identifiers on page 6-56. ■

Embed *default-catalog-name* is a value specification—a string literal or an SQL identifier—that specifies the catalog name. Enclose a string literal in single quotation marks (`'`); for example, `'mycatalog'`, where `mycatalog` is the name you choose. See Character String Literals on page 6-64. ■

Considerations for SET CATALOG

Scope of SET CATALOG

The default catalog you specify with SET CATALOG remains in effect until the end of the session or until you execute another SET CATALOG statement (or an equivalent SET SCHEMA statement).

If no SET CATALOG statement is in effect, NonStop SQL/MX determines the default catalog. For more information, see Object Naming on page 10-60.

Embed Use SET CATALOG to set a new default catalog for dynamic SQL statements. Use DECLARE CATALOG to set a new default catalog for static SQL statements. See DECLARE CATALOG Declaration on page 3-21. For more information, see the *SQL/MX Programming Manual for C and COBOL*. ■

MXCI Examples of SET CATALOG

- Set the default catalog name:

```
SET CATALOG mycatalog;
```

C Example of SET CATALOG

- Set the default catalog name with an SQL string literal:

  ```
  EXEC SQL SET CATALOG 'mycatalog';
  ```

- Set the default catalog name with an SQL identifier:

  ```
  EXEC SQL SET CATALOG mycatalog;
  ```

COBOL Example of SET CATALOG

- Set the default catalog name with an SQL string literal:

  ```
  EXEC SQL SET CATALOG 'mycatalog' END-EXEC.
  ```

- Set the default catalog name with an SQL identifier:

  ```
  EXEC SQL SET CATALOG mycatalog END-EXEC.
  ```

SET MPLOC Statement

The SET MPLOC statement sets the default NonStop operating system volume and subvolume for physical object names for the current SQL session.

Embed The SET MPLOC statement sets the default volume and subvolume for physical object names in all dynamic statements within the control flow scope of an embedded SQL program for the current SQL session. ■

SET MPLOC is an SQL/MX extension.

```
SET MPLOC [\node.]$volume.subvolume
```

`[\`*`node`*`.]`*`$volume`*`.`*`subvolume`*

is the fully qualified Guardian physical name of a subvolume. If you do not specify `\`*`node`*, the default is the Guardian node named in the =_DEFAULTS define.

Considerations for SET MPLOC

Scope of SET MPLOC

The default volume and subvolume you specify with SET MPLOC remains in effect until the end of the session or until you execute another SET MPLOC statement.

If no SET MPLOC statement is in effect, NonStop SQL/MX determines the default physical location. For more information, see Object Naming on page 10-60.

Embed Use SET MPLOC to set the default volume and subvolume for dynamic SQL statements. Use DECLARE MPLOC to set the default volume and subvolume for static SQL statements. See DECLARE MPLOC Declaration on page 3-29. For more information, see the *SQL/MX Programming Manual for C and COBOL*. ■

Examples of SET MPLOC

- Set the default volume and subvolume without setting the system:

  ```
  SET MPLOC $myvol.mysubvol;
  ```

- Set the default system, volume, and subvolume:

  ```
  SET MPLOC \aztec.$data06.part;
  ```

  If you then set the default volume and subvolume:

  ```
  SET MPLOC $data08.sales;
  ```

  the system will default to the system you previously set.

SET NAMETYPE Statement

Considerations for SET NAMETYPE
Examples of SET NAMETYPE

The SET NAMETYPE statement sets the NAMETYPE attribute value for the current SQL session.

Embed The SET NAMETYPE statement sets the NAMETYPE attribute for all dynamic statements within the control flow scope of an embedded SQL program for the current SQL session. ■

SET NAMETYPE is an SQL/MX extension.

```
SET NAMETYPE {ANSI | NSK}
```

`ANSI | NSK`

specifies whether the system assumes logical names (ANSI) or physical Guardian names (NSK) are used to reference SQL/MP database objects in SQL statements.

Considerations for SET NAMETYPE

Scope of SET NAMETYPE

The NAMETYPE attribute value you specify with SET NAMETYPE remains in effect until the end of the session or until you execute another SET NAMETYPE statement.

If no SET NAMETYPE statement is in effect, NonStop SQL/MX determines the default NAMETYPE attribute value. For more information, see Object Naming on page 10-60.

Embed Use SET NAMETYPE to set the NAMETYPE attribute for dynamic SQL statements. Use DECLARE NAMETYPE to set the NAMETYPE attribute for static SQL statements. See DECLARE NAMETYPE Declaration on page 3-32. For more information, see the *SQL/MX Programming Manual for C and COBOL*. ■

Examples of SET NAMETYPE

- Set the NAMETYPE attribute value to NSK:

```
SET NAMETYPE NSK;
```

SET SCHEMA Statement

The SET SCHEMA statement sets the default logical schema (and optionally the catalog) for unqualified object names for the current SQL session.

Embed The SET SCHEMA statement sets the default schema (and optionally the catalog) for unqualified object names in all dynamic statements within the control flow scope of an embedded SQL program for the current SQL session. ■

```
SET SCHEMA default-schema-name
```

default-schema-name

specifies the name of the schema and optionally the catalog. See Pseudocolumns on page 6-105.

MXCI *default-schema-name* is an SQL identifier. For example, you can use `MYSCHEMA` or `myschema` or a delimited identifier `"my schema"`. You can also specify both the catalog and schema as follows: `MYCATALOG.MYSCHEMA`. See Identifiers on page 6-56. ■

Embed *default-schema-name* is a value specification—a string literal or an SQL identifier—that specifies the default schema (and optionally the catalog). Enclose a string literal in single quotation marks (`'`); for example, `'sales'` for a default schema or `'samdbcat.sales'` for both a default catalog and schema. See Character String Literals on page 6-64. ■

Considerations for SET SCHEMA

Scope of SET SCHEMA

The default schema you specify with SET SCHEMA remains in effect until the end of the session or until you execute another SET SCHEMA statement.

If no SET SCHEMA statement is in effect, NonStop SQL/MX determines the default schema. For more information, see Object Naming on page 10-60.

Embed Use SET SCHEMA to set a new default schema for dynamic SQL statements. Use DECLARE SCHEMA to set a new default schema for static SQL statements. See DECLARE SCHEMA Declaration on page 3-33. For more information, see the *SQL/MX Programming Manual for C and COBOL*. ■

MXCI Examples of SET SCHEMA

- Set the default schema name:

```
SET SCHEMA myschema;
```

- Set the default catalog and schema name by specifying both:

```
SET SCHEMA mycatalog.myschema;
```

C Example of SET SCHEMA

- Set the default catalog and schema with an SQL string literal:

```
EXEC SQL SET SCHEMA 'prodcat.persnl';
```

- Set the default catalog and schema with an SQL identifier:

```
EXEC SQL SET SCHEMA prodcat.persnl;
```

COBOL Example of SET SCHEMA

- Set the default schema with an SQL string literal:

```
EXEC SQL SET SCHEMA 'prodcat.persnl' END-EXEC.
```

- Set the default schema with an SQL identifier:

```
EXEC SQL SET SCHEMA prodcat.persnl END-EXEC.
```

SET TABLE TIMEOUT Statement

Considerations for SET TABLE TIMEOUT
MXCI Examples of SET TABLE TIMEOUT
C Examples of SET TABLE TIMEOUT

The SET TABLE TIMEOUT statement sets a dynamic timeout value for a lock timeout or a stream timeout in the environment of the current session. The dynamic timeout value overrides the compiled static timeout value in the execution of subsequent DML statements.

You can use SET TABLE TIMEOUT from MXCI or in embedded SQL programs.

SET TABLE TIMEOUT is an SQL/MX extension.

To set the lock timeout

```
SET TABLE { * | table } TIMEOUT { value | RESET }
```

To set the stream timeout

```
SET TABLE * STREAM TIMEOUT { value | RESET }
```

`TABLE { * | table }`

specifies the name of the table. The table must exist in the user catalog before this statement is executed.

table can be any of:

- Guardian physical name of the form `[\node.][[$vol.]subvol.]filename`
- Three-part logical name of the form `[[catalog.]schema.]table`
- DEFINE name such as =CUSTOMER
- Host variable (If you use a host variable, you do not need to provide a PROTOTYPE clause.)

An asterisk (*) specifies all tables accessed in the current session. This option clears all previous dynamic timeout settings for specific tables in the current session.

Note. The *table* option is supported only for the lock timeout option. For the stream timeout option, you must use the asterisk (*) option.

Embed

If the table or DEFINE does not exist during explicit SQL/MX compilation, SET TABLE TIMEOUT returns an error. You must recompile the program if it has a missing table or DEFINE. For more information, see the *SQL/MX Programming Manual for C and COBOL*. ■

`TIMEOUT` *`value`*

specifies that *`value`* is for a lock timeout. If *`value`* elapses before a DML statement can acquire a lock on a table, the statement fails, and NonStop SQL/MX returns file-system error 73 (disk file or record is locked).

The *`value`* overrides any compiled values, such as those previously set by a CONTROL TABLE statement with the TIMEOUT option.

`STREAM TIMEOUT` *`value`*

specifies that *`value`* is for a stream timeout. A query that tries to access an empty stream waits until *`value`* elapses before NonStop SQL/MX returns:

```
*** ERROR[8006] The stream timed out, but the cursor is still
open.
```

The *`value`* overrides any compiled values, such as those previously set by a CONTROL QUERY DEFAULT statement with the STREAM_TIMEOUT option.

`value`

specifies the timeout value in hundredths of seconds.

Specify *`value`* as a:

- Numeric value (for example, `3000`)
- String with single quotation marks (for example, `'-1'`)
- **Embed** Host variable in an embedded SQL statement ■
- **MXCI** Parameter ■

The range is between -1 and 2147483519, expressed in hundredths of seconds. The value -1 represents an infinite timeout and directs NonStop SQL/MX not to time out.

A value of zero (0) directs NonStop SQL/MX not to wait. If a table lock cannot be acquired or if a stream is empty, NonStop SQL/MX immediately times out.

Note. Because of overhead processing by NonStop SQL/MX after a timeout occurs on a locked table, the actual time is usually a few seconds longer than *value*.

`RESET`

removes the dynamic timeout value (if set) for the specified table, resetting the timeout value to the static values set during explicit SQL/MX compilations. The RESET option with an asterisk resets the dynamic timeout value (lock or stream timeout, as specified) for all tables. The RESET option for a specific table does not override a dynamic timeout value that was set for all tables. (See MXCI Examples of SET TABLE TIMEOUT on page 2-374.)

Considerations for SET TABLE TIMEOUT

- The SET TABLE TIMEOUT statement does not perform any security checks on a table.
- A CONTROL statement is a directive that affects the compilation of subsequent DML statements but produces no executable code. A SET TABLE TIMEOUT statement, however, produces executable code and has no effect on the compilation of other statements.
- The SET TABLE TIMEOUT statement does not change the SQL/MX compilation defaults or CONTROL statement settings. A DML statement explicitly compiled after the execution of a SET TABLE TIMEOUT statement internally contains the static CONTROL statement timeout values, which are overridden by SET TABLE TIMEOUT.

Embed
- The SET TABLE TIMEOUT statement affects the run-time environment of an embedded SQL program. The explicit SQL/MX compilation defaults or CONTROL settings are not changed. DML statements compiled either before or after the execution of SET TABLE TIMEOUT still contain the same static timeout values in their code. ■

Embed
- The timeout values set by executing a SET TABLE TIMEOUT statement override the CONTROL directives that are in effect at the execution of subsequent DML statements. Therefore, you do not have to recompile a DML statement to change its timeout settings. The SET TABLE TIMEOUT statement also has a RESET option that clears previously set dynamic values, making the static values effective again. ■

Embed
- The timeout values set by a SET TABLE TIMEOUT statement are checked only when a DML statement is executed or when an SQL cursor is opened. Therefore, the statement has no effect on a cursor that is already open. ■

MXCI Examples of SET TABLE TIMEOUT

- Set the lock timeout value for all the tables to 30 seconds for the current session:

```
SET TABLE * TIMEOUT 3000;
```

- Set the lock timeout value for the CUSTOMER table to one minute:

```
SET TABLE customer TIMEOUT '6000';
SELECT custnum, custname FROM customer;
```

- Reset the timeout value for the CUSTOMER table to 30 seconds (set earlier for all tables):

```
SET TABLE customer TIMEOUT RESET;
```

- This statement has no effect; the PARTS table still uses the lock timeout value of 30 seconds (set earlier for all tables):

```
SET TABLE parts TIMEOUT RESET;
```

- Reset all the lock timeout settings. All tables will use the static lock timeout value specified by the system or by the CONTROL statement:

```
SET TABLE * TIMEOUT RESET;
```

C Examples of SET TABLE TIMEOUT

- Set the lock timeout value for all the tables to 30 seconds:

```
EXEC SQL SET TABLE * TIMEOUT 3000;
```

- Set the lock timeout value for the CUSTOMER table to 1 minute:

```
EXEC SQL SET TABLE CUSTOMER TIMEOUT '6000';
SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
      INTO :hv_custnum, :hv_custname,
           :hv_street, :hv_city, :hv_state, :hv_postcode
      FROM CUSTOMER
      WHERE CUSTNUM = :hv_custnum;
```

- This SET TABLE TIMEOUT statement has no effect because the cursor is already open:

```
EXEC SQL DECLARE mycursor CURSOR
  FOR SELECT custname FROM customer;
EXEC SQL OPEN mycursor;
EXEC SQL SET TABLE customer TIMEOUT '-1';
EXEC SQL FETCH mycursor INTO :customer_name;
```

- Reset the timeout value for the CUSTOMER table to 30 seconds (set earlier for all tables):

```
EXEC SQL SET TABLE CUSTOMER TIMEOUT RESET;
```

- This SET TABLE statement has no effect; the PARTS table still uses the lock timeout value of 30 seconds (set earlier for all tables):

```
EXEC SQL SET TABLE PARTS TIMEOUT RESET;
```

- Reset all the lock timeout settings. All tables will use the static lock timeout values set during explicit SQL/MX compilation:

```
EXEC SQL SET TABLE * TIMEOUT RESET;
```

- Set all streams to use a timeout of two minutes, and then reset the stream timeout to its original compile-time value:

```
EXEC SQL SET TABLE * STREAM TIMEOUT 12000;
EXEC SQL SELECT col1, col2 FROM STREAM(myqueue);

...
EXEC SQL SET TABLE * STREAM TIMEOUT RESET;
```

- Use a host variable to set a timeout value entered by a user:

```
/* Input timeout value into a host variable */
scanf("%ld",&hv_timeout);
```

```
/* Set timeout value for ORDERS table from host variable */
EXEC SQL SET TABLE orders TIMEOUT hv_timeout;
```

SET TRANSACTION Statement

Considerations for SET TRANSACTION
MXCI Examples of SET TRANSACTION
C Examples of SET TRANSACTION
COBOL Examples of SET TRANSACTION

The SET TRANSACTION statement is used to set attributes for the next transaction (and only the next transaction). The attributes are the isolation level, access mode, size of the diagnostics area, and whether to automatically commit changes made to the database. The isolation level and the access mode affect the degree of concurrency available for transactions.

```
SET TRANSACTION transaction-mode [,transaction-mode]...

transaction-mode is:
     isolation-level
   | access-mode

C/COBOL
   | diagnostics-size  ■

   | autocommit-option

   | autobegin-option


isolation-level is:
   ISOLATION LEVEL access-option

access-option is:
     READ UNCOMMITTED
   | READ COMMITTED
   | SERIALIZABLE
   | REPEATABLE READ

access-mode is:
   READ ONLY | READ WRITE

C/COBOL
diagnostics-size is:
   DIAGNOSTICS SIZE number-of-conditions ■

autocommit-option is:
   AUTOCOMMIT {ON | OFF}

autobegin-option is:
   AUTOBEGIN {ON | OFF}
```

transaction mode

is an option that can be set in a SET TRANSACTION statement. You cannot specify any of the options—isolation level, access mode, size of the diagnostics area, or autocommit—more than once within one SET TRANSACTION statement. You cannot use the AUTOCOMMIT option in combination with any other option.

isolation-level

specifies the level of data consistency defined for the transaction and the degree of concurrency the transaction has with other transactions that use the same data. The default isolation level of a transaction is determined according to the rules specified in Isolation Level on page 10-56.

access-mode

specifies the type of data access that the transaction requires, depending on whether changes are made to the database by the transaction.

If the *isolation-level* is READ UNCOMMITTED, you cannot specify READ WRITE. The default *access-mode* is READ ONLY, and you can specify only READ ONLY explicitly.

If the *isolation-level* is not READ UNCOMMITTED, you can specify either READ WRITE or READ ONLY explicitly. The default *access-mode* is READ WRITE.

See Transaction Access Modes on page 1-23.

C/COBOL

diagnostics-size

specifies the size of the diagnostics area (as an estimate of the number of expected conditions) used to return SQL query completion and exception condition information.

number-of-conditions is an exact numeric literal with zero scale. If the *diagnostics-size* is not set, it defaults to a system-defined value. ■

autocommit-option

specifies whether NonStop SQL/MX commits automatically or rolls back if an error occurs at the end of statement execution. This option applies to any statement for which the system initiates a transaction.

If this option is set to ON, NonStop SQL/MX automatically commits any changes or rolls back any changes made to the database at the end of statement execution. AUTOCOMMIT is ON by default at the start of an MXCI session or for embedded SQL in Java programs.

If this option is set to OFF, the current transaction remains active until the end of the MXCI session unless you explicitly COMMIT or ROLLBACK the transaction. AUTOCOMMIT is OFF by default for embedded SQL in C or COBOL programs.

Embed

If you exit a program without executing COMMIT or without setting AUTOCOMMIT ON, any uncommitted changes are automatically rolled back. ■

AUTOCOMMIT is an SQL/MX extension and cannot be used in combination with any other option.

autobegin-option

specifies that NonStop SQL/MX can start implicit transactions when the statement runs.

If this option is set to ON, NonStop SQL/MX automatically starts a transaction whenever a statement that requires a transaction is run.

Note. AUTOBEGIN is set to ON when a transaction is not available.

If this option is set to OFF, the transaction does not start automatically. You must explicitly start a transaction before running the statement. When a statement that requires a transaction is run, and there is no available transaction, Nonstop SQL/MX returns error 8877.

Note.

- AUTOBEGIN is a NonStop SQL/MX extension.
- AUTOBEGIN cannot be used in combination with any other option.
- AUTOBEGIN works only with embedded programs, namely, JDBC programs that use T2 driver and MXCI.
- AUTOBEGIN does not work with ODBC driver and JDBC T4 driver.

Considerations for SET TRANSACTION

Implicit Transactions

Most DML statements are transaction-initiating—the system automatically initiates a transaction when the statement begins executing.

The exceptions (statements that are not transaction-initiating) are:

- COMMIT, FETCH, ROLLBACK, and SET TRANSACTION
- DML statements operating on nonaudited tables
- DML statements executing under READ UNCOMMITTED access on audited tables
- C/COBOL The embedded-only SQL statements and declarations GET DIAGNOSTICS, BEGIN DECLARE SECTION, END DECLARE SECTION, and WHENEVER
- EXECUTE or EXECUTE IMMEDIATE, which are transaction-initiating only if the associated statement is transaction-initiating

Embed

- Cursor declarations (both static and dynamic) ■
- DECLARE CATALOG and DECLARE SCHEMA ■

In MXCI, the EXECUTE statement is transaction-initiating only if the statement that it executes is transaction-initiating.

Explicit Transactions

You can issue an explicit BEGIN WORK even if the autocommit option is on. The autocommit option is temporarily disabled until you explicitly issue COMMIT or ROLLBACK.

Degree of Concurrency

The SET TRANSACTION statement has an effect on the degree of concurrency available to the transaction. Concurrent processes take place within the same interval of time and share resources. The degree of concurrency available—that is, whether a process that requests access to data already being accessed is given access or placed in a wait queue—is affected by:

- The transaction access mode (READ ONLY or READ WRITE)
- The transaction isolation level (READ UNCOMMITTED, READ COMMITTED, SERIALIZABLE, or REPEATABLE READ)

Effect on Utilities

The SET TRANSACTION statement has no effect on the utility statements DUP, IMPORT, MODIFY TABLE, and PURGEDATA. The SET TRANSACTION statement does set attributes for transactions for UPDATE STATISTICS.

MXCI Examples of SET TRANSACTION

- Set the isolation level of a transaction that performs deletes, inserts, and updates:

```
SET TRANSACTION
  ISOLATION LEVEL SERIALIZABLE;
--- SQL operation complete.

BEGIN WORK;
--- SQL operation complete.

DELETE FROM persnl.employee
  WHERE empnum = 23;
--- 1 row(s) deleted.

INSERT INTO persnl.employee
  (empnum, first_name, last_name, deptnum, salary)
  VALUES (50, 'JERRY','HOWARD', 1000, 137000.00);
--- 1 row(s) inserted.

UPDATE persnl.dept
  SET manager = 50
  WHERE deptnum = 1000;
--- 1 row(s) updated.

COMMIT WORK;
--- SQL operation complete.
```

  This transaction uses SERIALIZABLE access (which provides maximum consistency but reduces concurrency). Therefore, you should execute it at a time when few users need concurrent access to the database. Locks acquired for SERIALIZABLE access are held until the changes made by these DELETE, INSERT, and UPDATE statements are committed.

C Examples of SET TRANSACTION

- Set the access option and isolation level for the next transaction within the program:

```
EXEC SQL SET TRANSACTION
  READ ONLY,
  ISOLATION LEVEL READ UNCOMMITTED;
```

COBOL Examples of SET TRANSACTION

- Set the access option and isolation level for the next transaction within the program:

```
EXEC SQL SET TRANSACTION
  READ ONLY,
  ISOLATION LEVEL READ UNCOMMITTED
END-EXEC.
```

SIGNAL SQLSTATE Statement

The SIGNAL statement is used with triggers. It allows a trigger execution to raise an exception that causes both the triggered and triggering statements to fail.

The SIGNAL statement sends an SQLSTATE and error text.

```
SIGNAL SQLSTATE quoted_sqlstate (quoted_string_expr);
```

quoted_sqlstate

is the five-digit SQLSTATE to be passed to SIGNAL.

quoted_string_expr

is a string expression.

Considerations for SIGNAL SQLSTATE

You can use the GET DIAGNOSTICS command to retrieve *quoted_string_expr* (as *message_text*) and *quoted_sqlstate*.

TABLE Statement

Considerations for TABLE
Examples of TABLE

The TABLE statement is equivalent to the query specification `SELECT * FROM` *`table`*.

```
TABLE table
```

`table`

names the user table or view.

Considerations for TABLE

Relationship to SELECT Statement

The result of the TABLE statement is one form of a *`simple-table`*, which is part of the definition of a table reference within a SELECT statement. See SELECT Statement on page 2-330.

Examples of TABLE

- This TABLE statement returns the same result as SELECT * FROM JOB:

```
TABLE JOB;

Job/Code  Job Description
--------  ------------------
     100  MANAGER
     200  PRODUCTION SUPV
     250  ASSEMBLER
     300  SALESREP
     400  SYSTEM ANALYST
     420  ENGINEER
     450  PROGRAMMER
     500  ACCOUNTANT
     600  ADMINISTRATOR
     900  SECRETARY

--- 10 row(s) selected.
```

UNLOCK TABLE Statement

Considerations for UNLOCK TABLE
Examples of UNLOCK TABLE

The UNLOCK TABLE statement releases locks owned by MXCI on a nonaudited SQL/MP table or on underlying nonaudited SQL/MP tables of a view. UNLOCK TABLE does not affect audited tables. Ending a transaction unlocks an audited table.

UNLOCK TABLE is an SQL/MX extension.

```
UNLOCK TABLE table
```

table

is the name of the table or view to unlock. See Database Object Names on page 6-13.

Considerations for UNLOCK TABLE

Authorization Requirements

To unlock a table, you must have authority to read the table. To unlock a view, you must have authority to read the view but not necessarily the tables underlying the view.

Examples of UNLOCK TABLE

- Lock and unlock a nonaudited table within an MXCI session:

```
LOCK TABLE persnl.job
  IN EXCLUSIVE MODE;
--- SQL operation complete.

DELETE FROM persnl.job
WHERE jobcode NOT IN
  (SELECT DISTINCT jobcode
   FROM persnl.employee);
--- 1 row(s) deleted.

UNLOCK TABLE persnl.job;
--- SQL operation complete.
```

UNREGISTER CATALOG Statement

The UNREGISTER CATALOG statement removes an empty SQL/MX catalog reference from a node.

```
UNREGISTER CATALOG catalog FROM \node [RESTRICT | CASCADE]
```

catalog

is the ANSI name of the target catalog. It must be visible on the local node.

\node

is the name of the target node, local or remote.

RESTRICT

specifies that only the reference for the named catalog will be removed. If that catalog is related to other catalogs, an error occurs.

RESTRICT is the default.

CASCADE

specifies that the references for the named catalog, and any catalogs that are directly or indirectly related to it, will be removed.

Considerations for UNREGISTER CATALOG

A catalog that is unregistered is no longer visible on the target node. The UNREGISTER CATALOG statement updates automatic catalog references to reflect that.

A catalog cannot be unregistered if any of the following is true:

- An object in the catalog is present on the target node, or depends on objects that are present on the target node.
- Definition schema tables for the catalog exist on the target node.

Authorization and Availability Requirements

To remove the catalog reference, you must be the user who created the catalog or be the super ID.

Example of UNREGISTER CATALOG

```
>> UNREGISTER CATALOG mycat FROM \nodex;
```

UPDATE Statement

[Considerations for UPDATE]
[MXCI Examples of UPDATE]
[C Examples of UPDATE]
[COBOL Examples of UPDATE]
[Publish/Subscribe Examples of DELETE]

The UPDATE statement is a DML statement that updates data in a row or rows in a table or updatable view. Updating rows in a view updates the rows in the table on which the view is based.

Starting with SQL/MX Release 3.2, self-referencing updates are supported. With this support, you can select the rows to update from the target table in a subquery.

Starting with SQL/MX release 3.2, you can update the primary key columns.

The two forms of the UPDATE statement are:

- Searched UPDATE—Updates rows whose selection depends on a search condition
- **Embed** Positioned UPDATE—Updates a single row determined by the cursor position. ■

For the searched UPDATE form, if there is no WHERE clause, all rows are updated in the table or view.

Use the positioned form of UPDATE only in embedded SQL programs. Use the searched form in MXCI or embedded SQL programs.

Searched UPDATE is:

```
Embed    [ ROWSET FOR INPUT SIZE rowset-size-in ] ■

         UPDATE table

Pub/Sub   | STREAM (table) [AFTER LAST ROW] ■

            SET set-clause [,set-clause ]...

Pub/Sub     [SET ON ROLLBACK set-roll-clause [,set-roll-clause]...] ■

            [WHERE search-condition | rowset-search-condition]
Embed       [[FOR] access-option ACCESS]

Pub/Sub   set-roll-clause is:
            column-name = expression ■  | rowset-expression ■
Embed
         access-option is:
            READ COMMITTED
          | SERIALIZABLE
          | REPEATABLE READ
          | SKIP CONFLICT
```

Embed

C/COBOL

Positioned UPDATE is:

```
UPDATE table
   SET set-clause [,set-clause]...

   WHERE CURRENT OF {cursor-name | ext-cursor-name} ■
 ■

set-clause is:
   column-name = {expression |rowset-expression | NULL}
```

Embed

`ROWSET FOR INPUT SIZE rowset-size-in`

Allowed only if you specify *rowset-search-condition* in the WHERE clause. *rowset-size-in* restricts the size of the input rowset to the specified size. If *rowset-size-in* is different from the allocated size for the rowset, NonStop SQL/MX uses the smaller of the two sizes and ignores the remaining entries in the larger rowset.

rowset-size-in must be an integer literal (exact numeric literal, dynamic parameter, or a host variable) whose type is unsigned short, signed short, unsigned long, or signed long in C and their corresponding equivalents in COBOL. If you do not specify *rowset-size-in*, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section of the embedded SQL program. ■

table

names the user table or view to update. *table* must be either a base table or an updatable view. To refer to a table or view, use one of these name types:

- Guardian physical name
- ANSI logical name
- DEFINE name

See Database Object Names on page 6-13.

Pub/Sub

`STREAM (table)`

updates a continuous data stream from the specified table. You cannot specify STREAM access for the UPDATE statement if it is not embedded as a table reference in a SELECT statement. See SELECT Statement on page 2-330.

`[AFTER LAST ROW]`

causes the stream to skip all existing rows in the table and update only rows that are published after the stream's cursor is opened. ■

set-clause

associates a value with a specific column in the table being updated. For each *set-clause*, the value of the specified target *column-name* is replaced by the value of the update source *expression* (or NULL). The data type of each target column must be compatible with the data type of its source value.

Embed

If you include a rowset search condition in the WHERE clause, you can use a rowset expression in *set-clause*, but it is not required. If the rowset sizes are different in the SET and WHERE clause, the smaller of the two sizes is used, and the remaining entries in the larger rowset are ignored. The rows selected by the *n*th condition in the rowset search condition are updated by the *n*th expression in the rowset expression in *set-clause*. See Rowset Search Condition on page 6-110. ■

column-name

names a column in *table* to update. You cannot qualify or repeat a column name. You cannot update the value of a column that is part of the primary key.

expression

is an SQL value expression that specifies a value for the column. The *expression* cannot contain an aggregate function defined on a column. The data type of *expression* must be compatible with the data type of *column-name*. A scalar subquery in *expression* cannot refer to the table being updated.

If *expression* refers to columns being updated, NonStop SQL/MX uses the original values to evaluate the expression and determine the new value.

See Expressions on page 6-41.

Embed

rowset-expression

is an array of SQL value expressions that specifies values for the column. A *rowset-expression* can appear in the SET clause only when a *rowset-search-condition* is present in the WHERE clause. When you use a *rowset-search-condition*, there are two alternatives for the *set-clause* expression:

- Scalar host variables only. In this case, all rows in the result table are updated with identical values, obtained by evaluating the scalar expression.
- Some array host variables. In this case, if the size of the array does not match the size of arrays used in the WHERE clause *search-condition*., the smaller value is used. All rows returned as a result of the first element in the search-condition array are updated using the value obtained by evaluating the first element in the *set-clause* array. All rows in the result table returned as a result of the second element in the *search-condition* array are updated using the second element in the *set-clause* array, and so on.

For details on using host variables and rowsets, see the *SQL/MX Programming Manual for C and COBOL*. ■

`NULL`

can also specify the value of the update source.

Pub/Sub `SET ON ROLLBACK` *`set-roll-clause`* `[,`*`set-roll-clause`*`]...`

causes one or more columns to be updated when the execution of the UPDATE statement causes its containing transaction to be rolled back.

`set-roll-clause`

sets the specified column to a particular value. For each *`set-roll-clause`*, the value of the specified target *`column-name`* is replaced by the value of the update source *`expression`*. The data type of each target column must be compatible with the data type of its source value.

Embed If you include a rowset search condition in the WHERE clause, you can use a rowset expression in *`set-roll-clause`*, but it is not required. If the rowset sizes are different in the SET and WHERE clause, the smaller of the two sizes is used, and the remaining entries in the larger rowset are ignored. The rows selected by the *n*th condition in the rowset search condition are updated by the *n*th expression in the rowset expression in *`set-roll-clause`*. ■

`column-name`

names a column in *`table`* to update. You cannot qualify or repeat a column name.

`expression`

is an SQL value expression that specifies a value for the column. *`expression`* cannot contain an aggregate function defined on a column. The data type of *`expression`* must be compatible with the data type of *`column-name`*. A scalar subquery in *`expression`* cannot refer to the table being updated.

If *`expression`* refers to columns being updated, NonStop SQL/MX uses the original values to evaluate the expression and determine the new value.

See Expressions on page 6-41.

`rowset-expression`

is an array of SQL value expressions that specifies values for the column. A *`rowset-expression`* can appear in the SET ON ROLLBACK clause only when a *`rowset-search-condition`* is present in the WHERE clause.

The rows returned by the nth element in the rowset-search-condition are updated by using the nth element in the rowset-expression. The rules described above for expression apply to each array element in the rowset-expression.

For details on using host variables and rowsets, see the *SQL/MX Programming Manual for C and COBOL*. ■

`WHERE` *`search-condition`*

specifies a *`search-condition`* that selects rows to update. Within the *`search-condition`*, columns being compared are also being updated in the table or view. See Search Condition on page 6-108.

If you do not specify a *`search-condition`*, all rows in the table or view are updated.

Do not use an UPDATE statement with a WHERE clause that contains a SELECT for the same table. Reading from and inserting into, updating in, or deleting from the same table generates an error. Use a positioned (WHERE CURRENT OF) UPDATE instead.

Embed `WHERE` *`rowset-search-condition`*

specifies an array of search conditions that selects rows to delete. The search conditions are applied successively and rows selected by each condition are updated before the next search condition is applied. Therefore, a single row can be updated multiple times. You can use a rowset expression in the set clause only if a rowset search condition is present. See Rowset Search Condition on page 6-110. ■

`[FOR]` *`access-option`* `ACCESS`

specifies the *`access-option`* required for data used in the evaluation of a search condition. See Data Consistency and Access Options on page 1-8.

`READ COMMITTED`

specifies that any data used in the evaluation of the search condition must be from committed rows.

`SERIALIZABLE | REPEATABLE READ`

specifies that the UPDATE statement and any concurrent process (accessing the same data) execute as if the statement and the other process had run serially rather than concurrently.

`SKIP CONFLICT`

enables transactions to skip rows locked in a conflicting mode by another transaction. The rows under consideration are the result of evaluating the search condition for the UPDATE statement. SKIP CONFLICT cannot be used in a SET TRANSACTION statement.

The default access option is the isolation level of the containing transaction, which is determined according to the rules specified in Isolation Level on page 10-56.

C/COBOL `WHERE CURRENT OF {cursor-name | ext-cursor-name}`

specifies the name of a cursor (or extended cursor) positioned at the row to update. If you specify *cursor-name* for an audited table or view, the UPDATE must execute within a transaction that also includes the FETCH for the row. Each column to be updated must appear in the FOR UPDATE clause of the cursor declaration. ■

For more information on searched and positioned UPDATE statements in embedded SQL programs, see the *SQL/MX Programming Manual for C and COBOL*.

Considerations for UPDATE

In a searched UPDATE, rows are updated in sequence.

Statement atomicity means that a statement will either complete or be rolled back, without having to rollback a business transaction that contains multiple statements. SQL/MX will try to undo any changes to the database as a result of an update in case a row cannot be updated.

There are some conditions where such an undo operation will cause an active transaction to be rolled back instead of just the statement. The following are some examples where the active transaction will be rolled back:

- Parallel inserts performed by ESPs
- VSBB inserts (either explicitly enforced by the CQD INSERT_VSBB set to ON or when chosen by the optimizer)
- CQD UPD_ABORT_ON_ERROR is set to ON to force transactions to be aborted. This CQD is supported to preserve the behavior of older releases.
- The underlying table has referential integrity constraints or triggers defined

For details, see Transaction Management on page 1-13.

Use the EXPLAIN statement to check whether transactions will be rolled back or if statement atomicity will be used. For details, see EXPLAIN Statement **on page 2-208**.

Authorization Requirements

UPDATE requires authority to read and write to the table or view being updated and authority to read any table or view specified in subqueries used in the search condition. A column of a view can be updated if its underlying column in the base table can be updated.

Transaction Initiation and Termination

The UPDATE statement will automatically initiate a transaction only if TRANSACTION AUTOBEGIN is set to ON. If a separate BEGIN WORK was issued, the UPDATE statement operates under that transaction.

The UPDATE statement will commit the transaction if TRANSACTION AUTOCOMMIT is set to ON. If AUTOCOMMIT is OFF, you must explicitly commit the transaction.

If a table is not audited, transactions do not apply.

Embed

Positioned UPDATE With AUTOCOMMIT

If you are using the positioned form of UPDATE, check that AUTOCOMMIT is set to OFF before you open a cursor. Otherwise, NonStop SQL/MX commits the transaction after each UPDATE statement and closes the cursor. Consequently, you might get rows fetched by your cursor that are part of different transactions. ■

Isolation Levels of Transactions and Access Options of Statements

The isolation level of an SQL/MX transaction defines the degree to which the operations on data within that transaction are affected by operations of concurrent transactions. When you specify access options for the DML statements within a transaction, you override the isolation level of the containing transaction. Each statement then executes with its individual access option.

Note. NonStop SQL/MX accepts SQL/MP keywords as synonyms for READ UNCOMMITTED, STABLE, and SERIALIZABLE.

You can explicitly set the isolation level of a transaction with the SET TRANSACTION statement. See SET TRANSACTION Statement on page 2-376. The default isolation level of a transaction is determined according to the rules specified in Isolation Level on page 10-56.

Embed

It is important to note that the SET TRANSACTION statement might cause a dynamic recompilation of the DML statements within the next transaction. Dynamic recompilation occurs if NonStop SQL/MX detects a change in the transaction mode at run time compared with the transaction mode at the time of static SQL compilation. To avoid dynamic recompilation because of a change in the transaction mode, consider specifying access options for individual DML statements instead of using SET TRANSACTION. ■

Conflicting Updates in Concurrent Applications

If you are using the READ COMMITTED isolation level within a transaction, your application can read different committed values for the same data at different times. Further, two concurrent applications can update (possibly in error) the same column in the same row.

In general, to avoid conflicting updates on a row, use the SERIALIZABLE isolation level. However, note that when you use SERIALIZABLE, you are limiting concurrent data access.

Requirements for Data in Row

Each row to be updated must satisfy the constraints of the table or underlying base table of the view. No column updates can occur unless all of these constraints are satisfied. (A table constraint is satisfied if the check condition is not false—that is, it is either true or has an unknown value.)

In addition, a candidate row from a view created with the WITH CHECK OPTION must satisfy the view selection criteria. The selection criteria are specified in the WHERE clause of the AS query-expression clause in the CREATE VIEW statement.

Reporting of Updates

When an UPDATE completes successfully, NonStop SQL/MX reports the number of times rows were updated during the operation.

Under certain conditions, updating a table with indexes can cause NonStop SQL/MX to update the same row more than once, causing the number of reported updates to be higher than the actual number of changed rows. However, both the data in the table and the number of reported updates are correct. This behavior occurs when all of these conditions are true:

- The optimizer chooses an alternate index as the access path.
- The index columns specified in WHERE `search-condition` are not changed by the update.
- Another column within the same index is updated to a higher value (if that column is stored in ascending order), or a lower value (if that column is stored in descending order).

When these conditions occur, the order of the index entries ensures that NonStop SQL/MX will encounter the same row (satisfying the same `search-condition)` at a later time during the processing of the table. The row is then updated again by using the same value or values.

For example, suppose that the index of MYTABLE consists of columns A and B, and the UPDATE statement is specified:

```
UPDATE MYTABLE
SET B = 20
WHERE A > 10;
```

If the contents of columns A and B are 11 and 12 respectively before the UPDATE, after the UPDATE NonStop SQL/MX will encounter the same row indexed by the values 11 and 20.

Updating Character Values

For a fixed-length character column, an update value shorter than the column length is padded with single-byte ASCII blanks (HEX20) to fill the column. If the update value is longer than the column length, string truncation of nonblank trailing characters returns an error, and the column is not updated.

For a variable-length character column, an update value is not padded; its length is the length of the value specified. As is the case for fixed length, if the update value is longer than the column length, string truncation of nonblank trailing characters returns an error, and the column is not updated.

In an SQL/MP entry-sequenced table, a value that updates a variable-length character column must be the same length as the value it replaces.

Audited and Nonaudited Tables

SQL/MX tables must be audited. You can run NonStop SQL/MX against nonaudited SQL/MP tables.

The Transaction Management Facility (TMF) product works only on audited tables, so a transaction does not protect operations on nonaudited tables. Nonaudited tables follow a different locking and error handling model than audited tables. Certain situations such as DML error occurrences or utility operations with DML operations can lead to inconsistent data within a nonaudited table or between a nonaudited table and its indices.

To avoid problems, do not run DDL or utility operations concurrently with DML operations on nonaudited tables. When you try to delete data in a nonaudited table with an index, NonStop SQL/MX returns an error.

Pub/Sub

SET ON ROLLBACK Considerations

The SET ON ROLLBACK expression is evaluated when each row is processed during execution of the UPDATE statement. The results of the evaluation are applied when and if the transaction is rolled back. This has two important implications:

- If the SET ON ROLLBACK expression generates an error (for example, a divide by zero or overflow error), the error is returned to the application when the UPDATE operation executes, regardless of whether the operation is rolled back.
- If an UPDATE operation is applied to a set of rows and an error is generated while executing the UPDATE operation, and the transaction is rolled back, the actions of the SET ON ROLLBACK clause apply only to the rows that were processed by the UPDATE operation before the error was generated. ■

Pub/Sub

SET ON ROLLBACK Restrictions

The table must be audited. The columns used in the SET ON ROLLBACK clause:

- Must be declared as NOT NULL.

- Cannot be part of a referential integrity constraint or be part of a secondary index.
- Cannot use the VARCHAR data type.
- Cannot be used in the primary key, clustering key, or partitioning key. ■

Embedded SELECT UPDATE Behavior

When you use a SELECT UPDATE statement to perform a searched UPDATE in an embedded statement, and more than one row satisfies the selection criteria, UPDATE may give unexpected results.

Suppose you have a table with two rows:

```
>>select * from =TAB1;

K1   K2   V3
---- ---- ----------------------------------------------------------------

0001 AAAA
0001 BBBB
```

If you perform this statement:

```
  EXEC SQL
       SELECT  *
       INTO :hv_k1
          , :hv_k2
          , :hv_v3
       FROM
          (UPDATE =TAB1
           SET v3 = 'ABCD'
           WHERE k1 ='0001'
           SKIP CONFLICT ACCESS
          )
       AS PIPO
       READ UNCOMMITTED ACCESS
       ;
```

Both rows would satisfy the selection criteria, so both rows could be updated. However, in this case it would be impossible to return the result, because NonStop SQL/MX can only return the values for one row from a statement like this.

Although NonStop SQL/MX cannot successfully update all rows and return the requested results, it does not return an error. Instead, only one row is updated, and the results for this single updated row are returned in the set of host variables.

If you execute the same SELECT UPDATE statement in MXCI NonStop SQL/MX returns these results:

```
>>select * from
+>(update =TAB1
+>set v3 = 'DCBA'
+>WHERE k1 ='0001'
+>SKIP CONFLICT ACCESS)
+>AS PIPO
+>READ UNCOMMITTED ACCESS
```

```
+>;

K1   K2    V3
---- ---- ----------------------------------------------------------------

0001 AAAA DCBA
0001 BBBB DCBA

--- 2 row(s) selected.
>>
```

NonStop SQL/MX updates both rows.

Primary key restrictions

UPDATES of primary key columns have the following restrictions:

- Cursor updates on primary key columns are not supported, SQL error 4118 is returned.
- Embedded UPDATES on primary key columns are not supported, SQL error 4198 is returned.
- SET ON ROLLBACK is not supported when:
  - A primary key column is updated, SQL error 4199 is returned. The following example returns SQL error 4199:

    ```
    >>create table t063t1 (a int not null, b int not null, c
    >>int not null, primary key(a));
    --- SQL operation complete.
    >>insert into t063t1 values (1,1,1), (2,2,2), (3,3,3),
    >>(4,4,4), (5,5,5);
    --- 5 row(s) inserted.
    >>update t063t1 set a = a+1 SET ON ROLLBACK c = 5;
    *** ERROR[4199] The SET ON ROLLBACK clause cannot be
    specified when clustering key columns are specified in the
    SET clause of an UPDATE statement.
    *** ERROR[8822] The statement was not prepared.
    ```

  - There is a primary key in the *set-roll-clause,* SQL error 4177 is returned. The following example returns SQL error 4177:

    ```
    >>update t063t1 set a = a+1 SET ON ROLLBACK a = 5;
    *** ERROR[4177] Update of Index Key column A is not permitted on
    rollback.
    *** ERROR[8822] The statement was not prepared.
    ```

- If a BEFORE trigger is defined on the primary key(s), SQL error 4033 is returned.

- Referential actions SET DEFAULT, SET NULL or SET CASCADE for ON UPDATE or ON DELETE rules are not supported, SQL error 4386 is returned.

For more information on the errors, see the *SQL/MX Messages Manual.*

MXCI Examples of UPDATE

- Update a single row of the ORDERS table that contains information about order number 200300 and change the delivery date:

```
UPDATE sales.orders
SET deliv_date = DATE '1998-05-02'
WHERE ordernum = 200300;
```

- Update several rows of the CUSTOMER table:

```
UPDATE sales.customer
SET credit = 'A1'
WHERE custnum IN (21, 3333, 324);
```

- Update all rows of the CUSTOMER table to the default credit 'C1':

```
UPDATE sales.customer
SET credit = 'C1';
```

- Update the salary of each employee working for all departments located in Chicago:

```
UPDATE persnl.employee
SET salary = salary * 1.1
WHERE deptnum IN
  (SELECT deptnum FROM persnl.dept
   WHERE location = 'CHICAGO');
```

  The subquery is evaluated for each row of the DEPT table and returns department numbers for departments located in Chicago.

- Suppose that you want to change the employee number of a manager of a department. Because EMPNUM is a primary key of the EMPLOYEE table, you must delete the employee's record and insert a record with the new number.

  You must also update the DEPT table to change the MANAGER column to the employee's new number. To ensure all your changes take place (or that none of them do), perform the operation as a transaction:

```
SET TRANSACTION
  ISOLATION LEVEL SERIALIZABLE;
--- SQL operation complete.

BEGIN WORK;
--- SQL operation complete.

DELETE FROM persnl.employee
  WHERE empnum = 23;
--- 1 row(s) deleted.
```

```
INSERT INTO persnl.employee
  (empnum, first_name, last_name, deptnum, salary)
  VALUES (50, 'JERRY','HOWARD', 1000, 137000.00);
--- 1 row(s) inserted.

UPDATE persnl.dept
  SET manager = 50
  WHERE deptnum = 1000;
--- 1 row(s) updated.

COMMIT WORK;
--- SQL operation complete.
```

This transaction uses SERIALIZABLE access, which provides maximum data consistency.

•

```
drop table test1;

create table test1(col1 int not null , col2 char(3),primary
key(col1) );

insert into test1 values ( 1, '100'), (2, '200');

--- 2 row(s) inserted.

prepare s1 from update test1 set col2 = '500' where col1 =
(select col1 from test1 where col1 > 1);

>>explain s1;

------------------------------------------------------------
----- PLAN SUMMARY
MODULE_NAME .............. DYNAMICALLY COMPILED
STATEMENT_NAME ........... S1
PLAN_ID .................. 212204692377308695
ROWS_OUT ................. 1
EST_TOTAL_COST ........... 0.03
STATEMENT ................ update test1
                           set col2 = '500'
                           where col1 = (select col1 from
test1 where col1 >
                             1);

------------------------------------------------------------
----- NODE LISTING
ROOT ======================================  SEQ_NO 11
ONLY CHILD 10
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0
EST_TOTAL_COST ........... 0.03
DESCRIPTION
  max_card_est ........... 1
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
```

```
  statement_index ........ 0
  olt_optimization ....... not used
  affinity_value  1,932,384,263
  est_memory_per_cpu .... 56
  upd_action_on_error .... savepoint
  xn_autoabort_interval    -1
  plan_version ....... 3,200
  self_referencing_update  forced_sort
  MXCI_PROCESS ........... ON
  SHOWCONTROL_UNEXTERNALI  OFF
  BLOCK_TO_PREVENT_HALLOW  ON
  select_list ............ %('500'), execution_count
..

>> execute s1;

--- 1 row(s) updated.
>>select * from test1;
COL1        COL2

----------- ----

          1 100

          2 500

--- 2 row(s) selected.
```

•

```
drop table test1;

create table test1(col1 int not null , col2 char(3),primary
key(col1) );
insert into test1 values ( 1, '100'), (2, '200');
--- 2 row(s) inserted.

prepare s1 from update test1 set col1 = 300 where col1 =
(select col1 from test1 where col1 > 1);
--- SQL command prepared.

>>explain s1;


-------------------------------------------------------------
----- PLAN SUMMARY

MODULE_NAME .............. DYNAMICALLY COMPILED
STATEMENT_NAME ........... S1
PLAN_ID .................. 212204692566543846
ROWS_OUT ................. 1
EST_TOTAL_COST ........... 0.06
STATEMENT ................ update test1
                           set col1 = 300
                           where col1 = (select col1 from
test1 where col1 >
                             1);
```

```
-------------------------------------------------------------
----- NODE LISTING

ROOT ======================================  SEQ_NO 15
ONLY CHILD 14
REQUESTS_IN .............. 1
ROWS_OUT ................. 1
EST_OPER_COST ............ 0
EST_TOTAL_COST ........... 0.06
DESCRIPTION
  max_card_est ........... 1
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  statement_index ........ 0
  olt_optimization ....... not used
  affinity_value  1,206,279,158
  est_memory_per_cpu .... 57
  upd_action_on_error .... savepoint
  xn_autoabort_interval    -1
  plan_version ....... 3,200
  self_referencing_update  forced_sort
  MXCI_PROCESS ........... ON
  SHOWCONTROL_UNEXTERNALI  OFF
  BLOCK_TO_PREVENT_HALLOW  ON
  select_list ............ %(300), %(300), execution_count

>>execute s1;

--- 1 row(s) updated.

>>select * from test1;

COL1         COL2

-----------  ----

          1  100

        300  200

--- 2 row(s) selected.
```

C Examples of UPDATE

- Reset the credit rating to the default value for all of the customers in the CUSTOMER table:

```
EXEC SQL UPDATE CUSTOMER SET CREDIT = DEFAULT;
```

- Use a loop to fetch and update by using a cursor:

```
...
CHAR SQLSTATE_OK[6]="00000";  /* variable declarations */
EXEC SQL BEGIN DECLARE SECTION;
  CHAR SQLSTATE[6];
  ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL FETCH cursor1 INTO SQL DESCRIPTOR 'out_sqlda';

while (strcmp(SQLSTATE, SQLSTATE_OK) == 0) {
  ...   /* retrieve and test values in descriptor area */
  EXEC SQL UPDATE CUSTOMER SET CREDIT = :new_default
           WHERE CURRENT OF cursor1;
  EXEC SQL FETCH cursor1 INTO SQL DESCRIPTOR 'out_sqlda';
}
...
```

COBOL Examples of UPDATE

- Reset the credit rating to the default value for all of the customers in the CUSTOMER table:

```
EXEC SQL UPDATE CUSTOMER SET CREDIT = DEFAULT END-EXEC.
```

- Use a loop to fetch and update by using a cursor:

```
  01 SQLSTATE-OK    PIC X(5) VALUE "00000".
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 SQLSTATE   PIC X(5).
        ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
     EXEC SQL FETCH cursor1
              INTO SQL DESCRIPTOR 'out_sqlda'
     END-EXEC.
     ...
     PERFORM UNTIL SQLSTATE NOT = SQLSTATE-OK
* Retrieve and test values in the descriptor area
       ...
       EXEC SQL UPDATE CUSTOMER SET CREDIT = :new-default
                WHERE CURRENT OF cursor1
       END-EXEC.
       EXEC SQL FETCH cursor1
              INTO SQL DESCRIPTOR 'out_sqlda'
       END-EXEC.
     END-PERFORM.
```

Publish/Subscribe Examples of UPDATE

Suppose that these SQL/MP tables and index (and the metadata mappings) have been created:

```
CREATE TABLE $db.dbtab.tab1 (a INT NOT NULL, b INT, c INT);
CREATE TABLE $db.dbtab.tab2 (a INT, b INT, c INT);
CREATE INDEX $db.dbtab.itab1 ON tab1(b, c);

CREATE SQLMP ALIAS cat.sch.tab1 $db.dbtab.tab1;
CREATE SQLMP ALIAS cat.sch.tab2 $db.dbtab.tab2;
```

- This example shows the SET ON ROLLBACK clause:

  ```
  SET SCHEMA cat.sch;

  UPDATE tab1
  SET b = b + 1
  SET ON ROLLBACK a = a + 1
  WHERE b < 10;
  ```

- This example shows the SKIP CONFLICT access:

  ```
  UPDATE tab1 SET a = a + 1
  FOR SKIP CONFLICT ACCESS;
  ```

UPDATE STATISTICS Statement

Considerations for UPDATE STATISTICS
Examples of UPDATE STATISTICS

The UPDATE STATISTICS statement updates the histogram statistics for one or more groups of columns within a table. These statistics are used to devise optimized access plans.

In addition to histogram statistics, UPDATE STATISTICS generates physical statistics (index level, non-empty block count, and EOF) for partitions of SQL/MX tables.

UPDATE STATISTICS is an SQL/MX extension.

```
UPDATE STATISTICS FOR TABLE table [CLEAR | on-clause]

on-clause is:
     ON column-group-list CLEAR
   | ON column-group-list [histogram-option]...

column-group-list is:
     column-list [,column-list]...
   | EVERY COLUMN [,column-list]...
   | EVERY KEY [,column-list]...

column-list for a single-column group is:
     column-name | (column-name)
   | column-name TO column-name
   | (column-name) TO (column-name)

column-list for a multicolumn group is:
     (column-name, column-name [,column-name]...)

histogram-option is:
     GENERATE n INTERVALS
   | SAMPLE [sample-option] [SET ROWCOUNT c ][sample-table-
clause]

sample-option is:
     [r ROWS]
   | RANDOM percent PERCENT [CLUSTERS OF blocks BLOCKS]
   | PERIODIC size ROWS EVERY period ROWS

sample-table-clause is:
USING SAMPLE TABLE {WITH PARTITIONS | sample-table-name}
```

table

names the table for which statistics are to be updated. To refer to a table, use one of these name types:

- Guardian physical name
- ANSI logical name

- DEFINE name

See Database Object Names on page 6-13.

`CLEAR`

deletes some or all histograms for the table *table*. Use this option when new applications no longer use certain histogram statistics.

If you do not specify *column-group-list*, all histograms for *table* are deleted.

If you specify *column-group-list*, only columns in the group list are deleted.

`ON` *column-group-list*

specifies one or more groups of columns, *column-group-list*, for which to generate histogram statistics with the option of clearing the histogram statistics. You must use the ON clause to generate statistics stored in histogram tables. If you omit it, physical statistics are generated for SQL/MX tables, and NonStop SQL/MX returns a warning message. See Using Statistics on page 2-406.

```
 column-list
| EVERY COLUMN [, column-list]
| EVERY KEY [, column-list]
```

specifies the ways in which *column-group-list* can be defined. The column list represents both a single-column group and a multicolumn group.

Single-column group:

```
 column-name | (column-name)
| column-name TO column-name
| (column-name) TO (column-name)
```

are the ways you can specify individual columns or a group of individual columns.

To generate statistics for individual columns, list each column. You have the option of listing each single column name within or without parentheses.

Multicolumn group:

```
(column-name, column-name [,column-name]...)
```

specifies a multicolumn group.

To generate multicolumn statistics, group a set of columns within parentheses, as shown. You cannot specify the name of a column more than once in the same group of columns.

One histogram is generated for each unique column group. Duplicate groups are ignored and processing continues. When you run UPDATE STATISTICS again for the same user table, the new data for that table replaces the data previously generated and stored in the table's histogram tables. Histograms of column groups not specified in the ON clause remain unchanged in histogram tables.

For more information about specifying columns, see Generating and Clearing Statistics for Columns on page 2-407.

`EVERY COLUMN`

The EVERY COLUMN keyword indicates that histogram statistics are to be generated for each individual column of *`table`* and any multicolumns that make up the primary key and indexes. For example, *`table`* has columns A, B, C, D defined, where A, B, C compose the primary key. In this case, the ON EVERY COLUMN option generates a single column histogram for columns A, B, C, D, and two multicolumn histograms of (A, B, C) and (A, B).

The EVERY COLUMN option does what EVERY KEY does, with additional statistics on the individual columns.

`EVERY KEY`

The EVERY KEY keyword indicates that histogram statistics are to be generated for columns that make up the primary key and indexes. For example, *`table`* has columns A, B, C, D defined. If the primary key comprises columns A, B, statistics are generated for (A, B), A and B. If the primary key comprises columns A, B, C, statistics are generated for (A,B,C), (A,B), A, B, C. If the primary key comprises columns A, B, C, D, statistics are generated for (A, B, C, D), (A, B, C), (A, B), and A, B, C, D.

`histogram-option`

`GENERATE` *`n`* `INTERVALS`

is an optional clause that specifies histograms are to be generated with approximately *n* number of intervals. The actual number of generated intervals might be more or less than the number *n*. Depending on the table's size and data distribution, each histogram should contain *n* intervals. NonStop SQL/MX attempts to distribute the rows evenly given the number of intervals.

The number *n* of intervals must be an integer between 1 and 10000 (1 < *n* < 10000). The interval number that you set is used for all column groups.

If you do not specify the number of intervals, a system default value is automatically provided based on the table size and other factors. It is recommended that you allow the system to determine the optimal number of intervals.

`SAMPLE [sample-option] [SET ROWCOUNT c ][sample-table-clause]`

is an optional clause that specifies that sampling is to be used to gather a subset of the data from the table. UPDATE STATISTICS uses a temporary table to store the sample results and generates histograms. See Histogram Table Properties on page 10-86 for details.

If you specify the `SAMPLE` clause without additional options, a row sample is used to read 2 percent of the rows in the table, with a maximum of 2 million rows. If you specify the `ROWCOUNT` option, NonStop SQL/MX reads 2 percent of *c*, with a maximum of 2 million rows.

If you do not specify the `SAMPLE` clause, *table* has fewer rows than specified, or the sample size is greater than the system limit. NonStop SQL/MX reads all rows from *table*.

See SAMPLE Clause on page 7-9.

sample-option

`[r ROWS]`

A row sample is used to read *r* rows from the table. The value *r* must be an integer that is greater than or equal to zero (*r* ≥ 0).

If you specify the `ROWCOUNT` clause, *r* must be less than or equal to *c* (*r* ≤ *c*). The percentage is determined by the equation r/c * 100.

`RANDOM percent PERCENT [CLUSTERS OF blocks BLOCKS]`

directs NonStop SQL/MX to choose rows randomly from the table. The value *percent* must be a value between zero and 100 (0 < *percent* ≤ 100). In addition, only the first four digits to the right of the decimal point are significant. For example, value 0.00001 is considered to be 0.0000, Value 1.23456 is considered to be 1.2345.

`CLUSTERS OF blocks BLOCKS`

specifies the number of blocks that compose the cluster. The value *block* must be an integer that is greater than or equal to zero (*blocks* ≥ 0).

`PERIODIC size ROWS EVERY period ROWS`

directs NonStop SQL/MX to choose the first *size* number of rows from each period of rows. The value *size* must be an integer that is greater than zero and less than or equal to the value *period*. (0 < *size* ≤ *period*). The size of the period is defined by the number of rows specified for *period*. The value *period* must be an integer that is greater than zero (*period* > 0).

`SET ROWCOUNT` *`c`*

is an optional clause that specifies the number of rows in the table. The value *c* must be an integer that is greater than or equal to zero ($c \geq 0$).

If the `ROWCOUNT` clause in not specified, NonStop SQL/MX determines the number of rows in the table either by estimation or `SELECT COUNT(*)`.

See SAMPLE Clause on page 7-9.

`sample-table-clause`

`USING SAMPLE TABLE WITH PARTITIONS`

directs SQL/MX to partition the temporary table. The temporary table is partitioned the same way as the base table on which the UPDATE STATISTICS command is run.

`USING SAMPLE TABLE` *`sample-table-name`*

directs SQL/MX to use the table specified by *`sample-table-name`* as the temporary table.

Note. The `sample-table-clause` is supported only for SQL/MX tables. It cannot be used with SQL/MP tables.

Considerations for UPDATE STATISTICS

Physical Statistics

Physical statistics (index level, nonempty block count, and EOF) are generated for UPDATE STATISTICS statements unless you use the `CLEAR` option.

Using Statistics

Use UPDATE STATISTICS to collect and save statistics on columns. The SQL compiler uses histogram statistics to determine the selectivity of predicates, indexes, and tables. Because selectivity directly influences the cost of access plans, regular collection of statistics increases the likelihood that NonStop SQL/MX will choose efficient access plans.

When a user table is changed, either by changing its data significantly or its definition, re-execute the UPDATE STATISTICS statement for the table.

Authorization and Locking

To run the UPDATE STATISTICS statement against SQL/MX tables, you must have the authority to read the user table for which statistics are generated. To run the UPDATE STATISTICS statement against SQL/MP tables, you must own the two histogram tables, or be the super ID, and have the authority to read the user table for which statistics are generated.

Because the histogram tables are registered in the schema (for SQL/MX tables) or catalog (for SQL/MP) of the primary partition of *table*, you must have the authority to read and write to this schema or catalog. Then, when the two histogram tables are created, you become the owner of the tables. See User Metadata Tables (UMD): Histogram Tables on page 10-85.

UPDATE STATISTICS momentarily locks the definition of the user table in the catalog during the operation but not the user table itself. The UPDATE STATISTICS statement uses READ UNCOMMITTED for the user table.

Transactions

Do not start a transaction before executing UPDATE STATISTICS because UPDATE STATISTICS runs under that transaction. The TMF auto abort time could be exceeded during the processing.

If you do not start a transaction for UPDATE STATISTICS, NonStop SQL/MX runs multiple transactions, breaking down the long transaction.

If the SQL/MP metadata files are locked, UPDATE STATISTICS tries three times to access them before reporting an error. Usually, metadata files are locked for short periods, and timeout errors do not occur. If the lock is held for a longer time, multiple retry attempts help to complete concurrent operations with minimum timeout interruption.

Generating and Clearing Statistics for Columns

To generate statistics for particular columns, name each column, or name the first and last columns of a sequence of columns in the table. For example, suppose that a table has consecutive columns CITY, STATE, ZIP. This list gives a few examples of possible options you can specify:

Single-Column Group	Single-Column Group Within Parentheses	Multicolumn Group
ON CITY, STATE, ZIP	ON (CITY),(STATE),(ZIP)	ON (CITY, STATE) *or* ON (CITY,STATE,ZIP)
ON CITY TO ZIP	ON (CITY) TO (ZIP)	
ON ZIP TO CITY	ON (ZIP) TO (CITY)	
ON CITY, STATE TO ZIP	ON (CITY), (STATE) TO (ZIP)	
ON CITY TO STATE, ZIP	ON (CITY) TO (STATE), (ZIP)	

The TO specification is useful when a table has many columns, and you want histograms on a subset of columns. Do not confuse (CITY) TO (ZIP) with (CITY, STATE, ZIP), which refers to a multicolumn histogram.

You can clear statistics in any combination of columns you specify, not necessarily with the *column-group-list* you used to create statistics. However, those statistics will remain until you clear them. For examples of SELECT statements to report on statistics, see Examples of Histogram Tables on page 10-92

Column Lists and Access Plans

Generate statistics for columns most often used in data access plans for a table—that is, the primary key, indexes defined on the table, and any other columns frequently referenced in predicates in WHERE or GROUP BY clauses of queries issued on the table. Use the EVERY COLUMN option to:

- generate histograms for every individual column or multicolumns that make up the primary key and indexes
- enable the optimizer to choose a better plan.

The EVERY KEY option generates histograms that make up the primary key and indexes.

If you often perform a GROUP BY over specific columns in a table, use multicolumn lists in the UPDATE STATISTICS statement (consisting of the columns in the GROUP BY clause) to generate histogram statistics that enable the optimizer to choose a better plan. Similarly, when a query joins two tables by two or more columns, multicolumn lists (consisting of the columns being joined) help the optimizer choose a better plan.

Sample Option

When you use the SAMPLE option, the UPDATE STATISTICS statement estimates the unique entry counts for each column for each histogram interval. The estimated unique entry counts tend to be more accurate for those columns that have fewer unique entries than the sampled number of rows. However, for columns that have been defined with the UNIQUE constraint, the unique entry count is always equal to the row count, regardless of the sample size. For more information about the unique entry count, see SAMPLE Clause on page 7-9.

If you specify the ROWCOUNT clause, use a value for *c* equal to the number of rows in the table. If you use a value that is less than or greater than the number of rows, results will not be accurate.

Sampling of Large Tables

Use the SAMPLE clause to reduce the run time for updating statistics of large tables:

- For tables with more than 2 million rows, use:

```
UPDATE STATISTICS FOR TABLE big_table ON EVERY COLUMN
  SAMPLE SET ROWCOUNT rowcount_big_table;
```

You can also specify groups of columns. This command uses the system default for sample size, which is 2 percent of the total number of rows in the table or 2 million rows, whichever is less.

- If `big_table` is highly skewed on certain columns (that is, a column has a large variance in the percentage of individual unique values), specify the sample size to be greater than the system default.

  Suppose that `big_table` has 50 million rows and you want to sample 10 percent of the rows. Use the command:

  ```
  UPDATE STATISTICS FOR TABLE big_table ON EVERY COLUMN
    SAMPLE RANDOM 10 PERCENT;
  ```

  You can also specify groups of columns.

Temporary Tables

Use the HIST_SCRATCH_VOL control query default to set the physical volume for UPDATE STATISTIC's temporary tables.

If you do not set this value, NonStop SQL/MX uses the default volume specified by the _DEFAULTS define and the current node for SQL/MX tables. If not specified, NonStop SQL/MX uses the volume of the table's primary partition for SQL/MP tables. The volume must be in the same node as the location of the catalog of the primary partition.

See the description of HIST_SCRATCH_VOL on page 10-55.

Using Sample Table with Partitions

While updating the statistics for SQL/MX tables, you can partition the temporary tables used by the UPDATE STATISTICS command. Use the USING SAMPLE TABLE WITH PARTITIONS clause to create a partitioned temporary table. When this clause is used, the temporary table is partitioned the same way as the base table for which the statistics are updated.

You can also create your own temporary table and specify it for the UPDATE STATISTICS command by using the USING SAMPLE TABLE *`sample-table-name`* clause. You can use this option to create a temporary table that has a different partitioning scheme from the default one. This includes:

- changing the number and nature of the partitions
- changing the key ranges
- controlling the disk layout

The table represented by *`sample-table-name`* must be a SQL/MX table. It must have the same column attributes as the base table—the columns must match in number, order, and data type. The table should not have any indexes, triggers, or constraints and it must be empty.

When you specify a *sample-table-name*, you must have ALL privileges on the temporary table and must own its schema or be the super ID.

Note. The USING SAMPLE TABLE clause is not supported with SQL/MP tables.

For more information on partitioned temporary tables, see the *SQL/MX Query Guide*.

When a SAMPLE clause is specified, the UPDATE STATISTICS command executes a SELECT statement with a corresponding SAMPLE clause and inserts the records into the temporary table. The sampling operation can be performed by either the SQL/MX Executor or the DP2. You can control this operation by using the ALLOW_DP2_ROW_SAMPLING default attribute. For more information on this attribute, see Default Attributes on page 10-39.

Managing SQL/MP Histograms

Before you drop an SQL/MP table, perform UPDATE STATISTICS with the CLEAR option. Otherwise, orphan histograms for that table are left on the system. However, if you drop an SQL/MP table before performing this step, use UPDATE STATISTICS with the CLEAR option to remove orphan tables:

1. Create a dummy table in the catalog where the primary partition of the table you dropped resided:

   ```
   CREATE TABLE trash (a INT);
   ```

2. Run UPDATE STATISTICS with the CLEAR option:

   ```
   UPDATE STATISTICS FOR TABLE trash CLEAR;
   ```

   The CLEAR option directs NonStop SQL/MX to remove histograms for table `trash`, and any orphaned histograms.

3. Drop the dummy table:

   ```
   DROP TABLE trash;
   ```

Histograms for SQL/MX tables are automatically deleted when the table is dropped.

Examples of UPDATE STATISTICS

For examples of histogram data, see Examples of Histogram Tables on page 10-92.

- This example generates four histograms for the columns jobcode, empnum, deptnum, and (empnum, deptnum) for the table EMPLOYEE. Depending on the table's size and data distribution, each histogram should contain 10 intervals.

  ```
  UPDATE STATISTICS FOR TABLE employee
    ON (jobcode),(empnum, deptnum)
    GENERATE 10 INTERVALS;

  --- SQL operation complete.
  ```

- This example generates histogram statistics using the ON EVERY COLUMN option for the table DEPT. This statement performs a full scan, and the NonStop SQL/MX determines the default number of intervals.

```
UPDATE STATISTICS FOR TABLE dept
  ON EVERY COLUMN;

--- SQL operation complete.
```

- This example generates statistics for a sample from table MAILINGS. The sample size is 7.3529 percent, and the number of rows in the table is 272,000.

```
UPDATE STATISTICS FOR TABLE mailings
  ON EVERY COLUMN
  SAMPLE RANDOM 7.3529 PERCENT CLUSTERS OF 1 BLOCKS
  SET ROWCOUNT 272000;
```

- Suppose that a construction company has an ADDRESS table of potential sites and a DEMOLITION_SITES table that contains some of the columns of the ADDRESS table. The primary key is ZIP. Join these two tables on two of the columns in common:

```
SELECT COUNT(AD.number), AD.street,
       AD.city, AD.zip, AD.state
  FROM address AD, demolition_sites DS
  WHERE AD.zip = DS.zip AND AD.type = DS.type
  GROUP BY AD.street, AD.city, AD.zip, AD.state;
```

  To generate statistics specific to this query, enter these statements:

```
UPDATE STATISTICS FOR TABLE address
  ON (street), (city), (state), (zip, type);

UPDATE STATISTICS FOR TABLE demolition_sites
  ON (zip, type);
```

- This example removes all histograms for table DEMOLITION_SITES:

```
UPDATE STATISTICS FOR TABLE demolition_sites CLEAR;
```

- This example selectively removes histograms for column STREET in table ADDRESS:

```
UPDATE STATISTICS FOR TABLE address ON street CLEAR;
```

- This example generates statistics for a sample from table MAILINGS. The sample size is 7.3529 percent and the number of rows in the table is 272,000. The records that are selected by sampling are stored in a temporary table, which is partitioned the same way as MAILINGS. The data in the temporary table is then used to generate the statistics.

```
UPDATE STATISITCS FOR TABLE MAILINGS
  ON EVERY COLUMN
  SAMPLE RANDOM 7.3529 PERCENT CLUSTERS OF 1 BLOCKS
  SET ROWCOUNT 272000
  USING SAMPLE TABLE WITH PARTITIONS;
```

- This example generates statistics for a sample from table MAILINGS. The sample size is 7.3529 percent and the number of rows in the table is 272,000. The records that are selected by sampling are stored in a temporary table, MY_SAMPLE_TABLE, which is specified by the user. The data in the temporary table is then used to generate the statistics.

```
UPDATE STATISITCS FOR TABLE MAILINGS
  ON EVERY COLUMN
  SAMPLE RANDOM 7.3529 PERCENT CLUSTERS OF 1 BLOCKS
  SET ROWCOUNT 272000
  USING SAMPLE TABLE MY_SAMPLE_TABLE;
```

For additional examples, see the *SQL/MX Query Guide*.

UPGRADE Utility

Considerations for UPGRADE
Example of UPGRADE

UPGRADE is a syntax-based utility command that can be executed from MXCI. UPGRADE transforms metadata from the existing version to the current schema version for the SQL/MX Software Version (MXV). The REPORTONLY option allows you to test if the operation can be executed without actually performing the operation.

```
UPGRADE ALL METADATA
    [ output-spec ]

UPGRADE ALL METADATA IN CATALOG catalog
    [ RESTRICT | CASCADE ]
    [ output-spec ]

output-spec is:
  [ log-to-spec ] [ REPORTONLY ]

log-to-spec is:
  { [ LOG TO ] OUTFILE oss-file [ CLEAR ] | LOG TO HOMETERM }
```

catalog

specifies the ANSI name of the catalog in which metadata is to be upgraded. There is no default for *catalog*.

RESTRICT

restricts the upgrade to metadata in the named catalog only.

CASCADE

extends the upgrade to the transitive closure of catalogs that are related to the specified catalog. If no option is specified, the default is CASCADE.

output-spec

corresponds to the output options.

Note. The UPGRADE utility is available only on systems running J06.11 and later J-series RVUs and H06.22 and later H-series RVUs.

Considerations for UPGRADE

Modes of Operation for UPGRADE

The following are two modes of operation for the UPGRADE utility:

- `ALL METADATA`

  This mode of operation upgrades all metadata that is visible on the system, including the metadata in the system catalog.

- `ALL METADATA IN CATALOG` *`catalog`*

  This mode of operation upgrades all the metadata in the named *catalog*. If the `CASCADE` option is specified, it affects the metadata in the catalogs that are related to the named *catalog*. This mode of operation is available in SQL/MX Release 3.2 and later releases.

Command Output for UPGRADE

The UPGRADE utility supports the following command output options:

- `REPORTONLY`

  If the `REPORTONLY` option is specified, only the initial error checking is performed and no upgrading takes place. If the `LOG TO` option is also specified, the list of affected schemas to be upgraded is written to the output file.

- `LOG TO`

  If the `LOG TO` option is specified, the command writes a log of its progress to either the specified *oss-file* or to the home terminal. If the `CLEAR` option is used and if *oss-file* is an existing disk file, *oss-file* is cleared before logging begins. Otherwise, the output is appended to the existing contents of *oss-file*. The following is the format of the first line of log output:

```
************** Time: <time> Process: <process>  Log opened **************
```

  The format enables you to recognize a log file easily. A command is rejected if it specifies an existing non-empty *oss-file* that is not a log file.

Log file messages correspond to the EMS event messages. Regardless of the `LOG TO` option, the UPGRADE utility will generate EMS events to the $0 primary collector that documents the progress of the command. For information about error messages, see the *SQL/MX Messages Manual*.

Error Conditions

The following are examples of the error conditions that might occur while executing the UPGRADE utility:

- An affected schema has a schema version that is higher than the target version.
- No schemas are affected by the operation.
- The `RESTRICT` option is specified or no option is specified, and one or more related catalogs exist.
- All of the following conditions are met:
  - The system catalog participates in the operation
  - Version 1200 schemas which do not participate in the operation exist on the system
- Concurrent UPGRADE operations are not supported.

Recovery of a Failed UPGRADE Utility

The RECOVER command is extended to allow recovery of a failed UPGRADE command. Starting with SQL/MX 3.2, the UPGRADE utility also allows recovery of an UPGRADE of metadata in an individual catalog.

```
RECOVER ALL METADATA
    [ RESUME | CANCEL ]
    [ output-spec ]

RECOVER ALL METADATA IN CATALOG catalog
    [ RESUME | CANCEL ]
    [ output-spec ]
```

`catalog`

specifies the ANSI name of the catalog that was specified in the original UPGRADE command.

`RESUME`

enables you to continue with the processing of the original command, starting at the point of interruption.

`CANCEL`

enables you to revert the changes made by the original command, thereby returning the database to its original state. The default value is CANCEL.

`output-spec`

is the same as for the UPGRADE operation.

The RECOVER command must use the same mode of operation as the original UPGRADE command:

- If the original UPGRADE used the ALL METADATA mode of operation, then the RECOVER must also use the ALL METADATA mode of operation.
- If the original UPGRADE used the ALL METADATA IN CATALOG catalog mode of operation, then the RECOVER must also use the ALL METADATA IN CATALOG catalog mode of operation, and must specify the same catalog name.

In a distributed environment, the RECOVER command must be issued from the same system where the original UPGRADE command was executed.

Error Conditions

The following are examples of the error conditions that might occur while executing the RECOVER command:

- An involved node has an incompatible version (because the version of the node was modified between the time of the original operation and the time of recovery)
- No corresponding UPGRADE or DOWNGRADE operation is recorded
- The original command is still active

Note. The RECOVER command does not need CASCADE option as it automatically recovers the metadata for those catalogs that were affected by the original UPGRADE command.

Example of UPGRADE

This example transforms all the metadata to the `/usr/dbadmin/upgradeLog` file:

```
UPGRADE ALL METADATA LOG TO OUTFILE /usr/dbadmin/upgradeLog
CLEAR;
```

The following is an excerpt from the output file.

```
*************** Time: <time> Process: <process>  Log opened
***************
The UPGRADE ALL METADATA has started
                       ...
Schema XCAT.ASCH will be upgraded from version 1200 to version 3000
Schema YCAT.ASCH will be upgraded from version 1200 to version 3000
Schema YCAT.ZSCH will be upgraded from version 1200 to version 3000
Schema ZCAT.SCH1 will be upgraded from version 1200 to version 3000
Schema ZCAT.SCH2 will be upgraded from version 1200 to version 3000
                       ...
Creating version 3000 definition schema for catalog XCAT
Upgrading version 1200 metadata to version 3000 for affected schemas in
catalog XCAT
Set schema version to 3000 for XCAT.ASCH
Remove XCAT.DEFINITION_SCHEMA_VERSION_1200
Schema XCAT.ASCH has been upgraded from version 1200 to version 3000.
Creating version 3000 definition schema for catalog YCAT
                       ...
Set schema version to 3000 for ZCAT.SCH1
Set schema version to 3000 for ZCAT.SCH2
Schema ZCAT.SCH1 has been upgraded from version 1200 to version 3000.
Schema ZCAT.SCH2 has been upgraded from version 1200 to version 3000.
Remove ZCAT.DEFINITION_SCHEMA_VERSION_1200
The UPGRADE ALL METADATA has completed
```

Note. The date-time-processid prefix of each line and the output for schemas in the system catalog are not displayed in the output file. Also, source definition schemas are removed as part of the operation.

VALUES Statement

The VALUES statement starts with the VALUES keyword followed by a sequence of row value constructors, each of which is enclosed in parentheses. It displays the results of the evaluation of the expressions and the results of row subqueries within the row value constructors.

```
VALUES (row-value-constructor) [,(row-value-constructor)]...

row-value-constructor is:
     row-subquery
   | {expression | NULL} [,{expression | NULL}]...
```

row-value-constructor

specifies a list of expressions (or NULL) or a row subquery (a subquery that returns a single row of column values). An operand of an expression cannot reference a column (except when the operand is a scalar subquery returning a single column value in its result table).

The use of NULL as an element of a *row-value-constructor* is an SQL/MX extension.

The results of the evaluation of the expressions and the results of the row subqueries in the row value constructors must have compatible data types.

Considerations for VALUES

Relationship to SELECT Statement

The result of the VALUES statement is one form of a *simple-table*, which is part of the definition of a table reference within a SELECT statement. See SELECT Statement on page 2-330.

Examples of VALUES

- This VALUES statement displays the results of the expressions in the list:

```
VALUES (1,2,3);

(EXPR)  (EXPR)  (EXPR)
------  ------  ------
     1       2       3

--- 1 row(s) selected.
```

- This VALUES statement displays the results of the expressions and the row subquery in the lists:

```
VALUES ('a','b',UPSHIFT('c')),
  ((SELECT jobdesc FROM job WHERE jobcode=300),'d',NULL);

(EXPR)              (EXPR)  (EXPR)
------------------  ------  ------
a                   b       C
SALESREP            d            ?

--- 2 row(s) selected.
```

3 Embedded-Only SQL/MX Statements

This section describes the syntax and semantics of NonStop SQL/MX statements that you can embed only in programs written in C, C++, COBOL, or Java.

In NonStop SQL/MX Release 2.x, mixing embedded SQL calls to NonStop SQL/MP and NonStop SQL/MX from the same application process is not supported.

You cannot run these statements, or specific forms of these statements, in MXCI:

	DECLARE SCHEMA Declaration on page 3-33	Sets default schema for unqualified object names in static SQL statements within a compilation unit.
C/COBOL	DESCRIBE Statement on page 3-34	Uses an SQLDA to return descriptions of output variables (usually SELECT columns) and input parameters for a prepared statement.
C/COBOL	END DECLARE SECTION Declaration on page 3-37	Designates the end of a Declare Section.
C/COBOL	EXEC SQL Directive on page 3-38	Begins an embedded SQL statement or declaration.
C/COBOL	EXECUTE IMMEDIATE Statement on page 3-39	Prepares (compiles) and executes a dynamic SQL statement.
	FETCH Statement on page 3-40	Retrieves a row using a cursor.
C/COBOL	GET DESCRIPTOR Statement on page 3-46	Retrieves information from an SQLDA.
C/COBOL	GET DIAGNOSTICS Statement on page 3-55	Returns diagnostic information about the most recently executed SQL statement.
C/COBOL	IF Statement on page 3-61	Compound statement that provides conditional execution based on the truth value of a conditional expression.
C/COBOL	INVOKE Directive on page 3-64	Generates a structure description of a table or view.
	MODULE Directive on page 3-70	Specifies the name of an embedded SQL module for the preprocessor.
C/COBOL	OPEN Statement on page 3-72	Opens a cursor.
	SET (Assignment) Statement on page 3-76	Assigns a value to a host variable so that subsequent statements in the containing compound statement can reference and use the value of that host variable.
C/COBOL	SET DESCRIPTOR Statement on page 3-78	Modifies information in an SQLDA.
C/COBOL	WHENEVER Declaration on page 3-86	Unloads the module files.
C/COBOL	WHENEVER Declaration on page 3-86	Generates code that checks SQL statement execution for errors and the end no-data condition and specifies an action to take.

For more information on how to embed SQL/MX statements in C or COBOL programs, see the *SQL/MX Programming Manual for C and COBOL*.

ALLOCATE CURSOR Statement

Considerations for ALLOCATE CURSOR
C Examples of ALLOCATE CURSOR
COBOL Examples of ALLOCATE CURSOR

C/COBOL The ALLOCATE CURSOR statement is a dynamic SQL statement used to define an extended cursor based on a statement already prepared for the cursor specification. It allows applications to dynamically create an unlimited number of cursors.

Use ALLOCATE CURSOR only in embedded SQL programs in C or COBOL.

Pub/Sub

```
ALLOCATE ext-cursor-name
  CURSOR [WITH HOLD | WITHOUT HOLD]
  FOR ext-statement-name

ext-cursor-name is:
   [GLOBAL | LOCAL] value-specification

ext-statement-name is:
   [GLOBAL | LOCAL] value-specification
```

ext-cursor-name

is a *value-specification*—a host variable with character data type. When ALLOCATE CURSOR executes, the content of the host variable gives the name of the cursor. The maximum length of a cursor name is 128 characters.

Pub/Sub WITH HOLD | WITHOUT HOLD

specifies whether an application keeps cursors open (WITH) across transaction boundaries. The default is WITHOUT HOLD. You can use the WITH HOLD clause only with Publish/Subscribe. ■

GLOBAL | LOCAL

specifies scope. The default setting is LOCAL. The scope of a GLOBAL cursor or statement name is the SQL session. The scope of a LOCAL cursor or statement name is the module or compilation unit in which ALLOCATE CURSOR appears.

ext-statement-name

is a *value-specification*—a host variable with character data type. When ALLOCATE CURSOR executes, the content of the host variable must identify a statement previously prepared within the scope of ALLOCATE CURSOR. The prepared statement must be a cursor specification.

When host variables are used for the *ext-cursor-name* and the *ext-statement-name* in the DECLARE CURSOR statement, the ALLOCATE

CURSOR statement is functionally equivalent to the DECLARE CURSOR statement.

Considerations for ALLOCATE CURSOR

Cursor Names

You cannot have more than one cursor allocated with the same name within the same scope. For example, this sequence from a C program is not valid:

```
strcpy(extcur1,"CURSOR1");
EXEC SQL ALLOCATE :extcur1 CURSOR FOR :stmt;
strcpy(extcur2,"CURSOR1");
EXEC SQL ALLOCATE :extcur2 CURSOR FOR :stmt;
```

The second ALLOCATE CURSOR fails because CURSOR1 has already been allocated.

Using Extended Dynamic Cursors

The name of an extended dynamic cursor is not known until run time. Therefore, you can allocate new cursors as you need them.

However, you must have prepared a cursor specification and stored the name of the prepared cursor specification in a host variable before ALLOCATE CURSOR executes.

Pub/Sub

WITH HOLD

You can use holdable cursors only for SELECT statements that use the Publish/Subscribe stream access mode or an embedded UPDATE or embedded DELETE. ■

C Examples of ALLOCATE CURSOR

This example uses extended cursor and statement names in the PREPARE and ALLOCATE CURSOR statements:

```
...
scanf("%s", in_curspec);
...
EXEC SQL PREPARE :curspec FROM :in_curspec;
...
EXEC SQL ALLOCATE :extcur CURSOR FOR :curspec;
...
```

COBOL Examples of ALLOCATE CURSOR

This example uses extended cursor and statement names in the PREPARE and ALLOCATE CURSOR statements:

```
...
ACCEPT in-curspec.
...
EXEC SQL PREPARE :curspec FROM :in-curspec END-EXEC.
...
EXEC SQL ALLOCATE :extcur CURSOR FOR :curspec END-EXEC.
...
```

Publish/Subscribe Examples of ALLOCATE CURSOR

This example uses the WITH HOLD clause with ALLOCATE CURSOR:

```
...
EXEC SQL
  ALLOCATE :ext_hold_stmthold CURSOR WITH HOLD FOR
    :szHoldableStatementName;

EXEC SQL BEGIN WORK;
EXEC SQL OPEN :ext_hold_stmthold;
for( ;; ) {
          EXEC SQL FETCH :ext_hold_stmthold INTO :hv;
          EXEC SQL COMMIT;
          EXEC SQL BEGIN WORK;
}
```

ALLOCATE DESCRIPTOR Statement

Considerations for ALLOCATE DESCRIPTOR
C Examples of ALLOCATE DESCRIPTOR
COBOL Examples of ALLOCATE DESCRIPTOR

C/COBOL The ALLOCATE DESCRIPTOR statement allocates a named SQL descriptor area used for storing information necessary for the execution of dynamic SQL statements.

Use ALLOCATE DESCRIPTOR only in embedded SQL programs in C or COBOL.

```
ALLOCATE DESCRIPTOR descriptor-name WITH MAX occurrences

descriptor-name is:
   [GLOBAL | LOCAL] value-specification
```

descriptor-name

is a *value-specification*—a character literal or host variable with character data type. When ALLOCATE DESCRIPTOR executes, the content of the host variable (if used) gives the name of the descriptor area.

GLOBAL | LOCAL

specifies the scope of the allocated descriptor area. The default setting is LOCAL. A GLOBAL descriptor area is available to the SQL session. A LOCAL descriptor area is available only to the module or compilation unit in which it was allocated.

WITH MAX *occurrences*

specifies the maximum number of items in the descriptor area. *occurrences* must be a host variable. The specified area must be large enough to store information for as many parameters as you are using in your dynamic SQL statements. The data type of *occurrences* must be exact numeric with scale 0 and a value of 1 or greater.

Considerations for ALLOCATE DESCRIPTOR

You should code the ALLOCATE DESCRIPTOR statement before the PREPARE statement for the input descriptor. For example:

```
EXEC SQL ALLOCATE DESCRIPTOR 'in_desc' WITH MAX :desc_max;

printf("SQLCODE after allocate descriptor - 1 is %d\n",
    SQLCODE);

strncpy(insert_buf, " ", sizeof(insert_buf));

strcpy(insert_buf, "insert into a5tab1 (select * from
 a5tab0 where double1 between ? and double2/2);");
```

```
hvdouble1 = 1.0E-76;

EXEC SQL PREPARE insert_q FROM :insert_buf;
   printf("SQLCODE after prepare insert_q - 1 is %d\n",
SQLCODE);

Exec SQL execute insert_q using :hvdouble1;
   printf("SQLCODE after insert - 3 is %d \n", SQLCODE);
```

Defining Values in the Descriptor Area

All values in all items of the descriptor area are initially undefined. To define values, use a DESCRIBE statement or explicitly set values with a SET DESCRIPTOR statement.

Descriptor Names

You cannot have more than one descriptor allocated with the same name at the same time within the same scope. For example, this sequence from a C program is not valid:

```
strcpy(descname1,"SQLDA1");
desc_max1 = 2;
EXEC SQL ALLOCATE DESCRIPTOR :descname1 WITH MAX :desc_max1;
strcpy(descname2,"SQLDA1");
desc_max2 = 3;
EXEC SQL ALLOCATE DESCRIPTOR :descname2 WITH MAX :desc_max2;
```

The second ALLOCATE DESCRIPTOR fails because SQLDA1 has already been allocated.

C Examples of ALLOCATE DESCRIPTOR

- This example uses an SQL string literal as the descriptor name:

```
desc_max = 1;
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc_max;
```

- This example uses a host variable as the descriptor name:

```
...
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR desc_name[20];
long    desc_max;
...
EXEC SQL END DECLARE SECTION;
...
strcpy(desc_name, "in_sqlda");
desc_max = 1;
EXEC SQL ALLOCATE DESCRIPTOR :desc_name WITH MAX :desc_max;
...
```

COBOL Examples of ALLOCATE DESCRIPTOR

- This example uses an SQL string literal as the descriptor name:

```
MOVE 1 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda'
           WITH MAX :desc-max
END-EXEC.
```

- This example uses a host variable as the descriptor name:

```
...
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 desc-name   PIC X(20).
01 desc-max    S9(9) comp.
...
   EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE "in_sqlda" TO desc-name.
MOVE 1 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR :desc-name
         WITH MAX :desc-max
END-EXEC.
...
```

BEGIN DECLARE SECTION Declaration

C/COBOL

BEGIN DECLARE SECTION is a preprocessor directive that begins SQL declarations in a host program. SQL declarations are used to define host variables to be used in SQL/MX statements—for example, to transfer data to and from a database.

Use BEGIN DECLARE SECTION only in embedded SQL programs in C or COBOL.

```
BEGIN DECLARE SECTION
```

See END DECLARE SECTION Declaration on page 3-37.

C Examples of BEGIN DECLARE SECTION

- This example shows a declaration section:

```
EXEC SQL BEGIN DECLARE SECTION;
  SHORT length;
  CHAR  data[10];
EXEC SQL END DECLARE SECTION;
```

- This example shows a declaration section that uses an INVOKE directive to declare a structure template of a table:

```
EXEC SQL BEGIN DECLARE SECTION;
  EXEC SQL INVOKE SALES.PARTS;
EXEC SQL END DECLARE SECTION;
```

C++ Examples of BEGIN DECLARE SECTION

- This example shows a declaration section within a class. Member functions using these host variables must be defined within the visible scope of the class.

```
class jobsql {
// Class member host variables
EXEC SQL BEGIN DECLARE SECTION;
  short length;
  VARCHAR data[19];
EXEC SQL END DECLARE SECTION;
public:
...
  }
};  // End of jobsql class definition
```

COBOL Examples of BEGIN DECLARE SECTION

- This example shows a declaration section:

```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 length   pic 9(4)comp.
01 data     pic x(10).
   EXEC SQL END DECLARE SECTION END-EXEC.
```

- This example shows a declaration section that uses an INVOKE directive to declare a record description of a table:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  EXEC SQL INVOKE SALES.PARTS END-EXEC.
EXEC SQL END DECLARE SECTION END-EXEC.
```

CLOSE Statement

C/COBOL

The CLOSE statement closes a cursor in a host program and releases the result table established by the OPEN statement for the cursor. The COMMIT WORK statement or ROLLBACK WORK statement also closes all open cursors in a host program and releases all result tables.

In dynamic SQL, the cursor name is provided at execution time. Otherwise, there is no difference in the static and dynamic forms of CLOSE.

Use CLOSE only in embedded SQL programs in C or COBOL.

```
CLOSE {cursor-name  | ext-cursor-name}

ext-cursor-name is:
   [GLOBAL | LOCAL] value-specification
```

cursor-name

is an SQL identifier—the name of an open cursor. See Identifiers on page 6-56.

GLOBAL | LOCAL

specifies scope. The default setting is LOCAL. The scope of a GLOBAL cursor is the SQL session. The scope of a LOCAL cursor is the module or compilation unit in which CLOSE appears.

ext-cursor-name

is a *value-specification*—a character literal or a host variable with character data type. When CLOSE executes, the content of the value specification (if a host variable) gives the name of the cursor.

Considerations for CLOSE

Scope of CLOSE

The module or compilation unit that contains the CLOSE statement also has a DECLARE CURSOR statement that uses the same cursor name. The cursor name in the CLOSE statement is associated with the cursor specification in this DECLARE CURSOR.

Reusing a Cursor

After CLOSE executes, the result table for the cursor (the output that results from the execution of the SELECT that specifies the cursor) no longer exists. To use the same cursor again, you must reopen it with an OPEN statement.

Effect on Locks

Closing a cursor does not affect locks. Locks on audited tables are released when the containing transaction completes or aborts; locks on nonaudited tables must be released with UNLOCK TABLE.

Using Extended Dynamic Cursors

The name of an extended dynamic cursor is not known until run time. When CLOSE executes, the name must identify an open cursor within the same scope.

C Examples of CLOSE

- Declare and open a cursor, fetch a row of retrieved data, and then close the cursor. Note that in an actual program you would include processing the data in the host variables `hostvar1`, `hostvar2`, and `hostvar3`, and looping back to fetch the next row provided by the cursor.

```
...
EXEC SQL DECLARE cursor1 CURSOR FOR
     SELECT COL1, COL2, COL3 FROM SALES.PARTS
     WHERE  COL1  >= :hostvar1
     ORDER BY COL1
     READ UNCOMMITTED ACCESS;
...  /* Initialize value of hostvar1 */
EXEC SQL OPEN cursor1;
...
EXEC SQL FETCH cursor1 INTO :hostvar1, :hostvar2, :hostvar3;
...
EXEC SQL CLOSE cursor1;
```

- This example uses extended cursor and statement names in the PREPARE, ALLOCATE CURSOR, OPEN, and CLOSE statements.

```
...
scanf("%s", in_curspec);
...
EXEC SQL PREPARE :curspec FROM :in_curspec;
...
EXEC SQL ALLOCATE :extcur CURSOR FOR :curspec;
...
EXEC SQL OPEN :extcur;
/* Process using the extended dynamic cursor. */
...
EXEC SQL CLOSE :extcur;
```

COBOL Examples of CLOSE

- Declare and open a cursor, fetch a row of retrieved data, then close the cursor. Note that in an actual program you would include processing the data in the host variables `hostvar1`, `hostvar2`, and `hostvar3`, and looping back to fetch the next row provided by the cursor.

```
     ...
     EXEC SQL DECLARE cursor1 CURSOR FOR
       SELECT COL1, COL2, COL3 FROM SALES.PARTS
       WHERE  COL1  >= :hostvar1
       ORDER BY COL1
       READ UNCOMMITTED ACCESS END-EXEC.
* Initialize value of hostvar1
     ...
     EXEC SQL OPEN cursor1 END-EXEC.
     ...
     EXEC SQL FETCH cursor1
        INTO :hostvar1, :hostvar2, :hostvar3 END-EXEC.
     ...
     EXEC SQL CLOSE cursor1 END-EXEC.
```

- This example uses extended cursor and statement names in the PREPARE, ALLOCATE CURSOR, OPEN, and CLOSE statements.

```
     ...
     ACCEPT in-curspec.
     ...
     EXEC SQL PREPARE :curspec FROM :in-curspec END-EXEC.
     ...
     EXEC SQL ALLOCATE :extcur CURSOR FOR :curspec END-EXEC.
     ...
     EXEC SQL OPEN :extcur END-EXEC.
* Process using the extended dynamic cursor.
     ...
     EXEC SQL CLOSE :extcur END-EXEC.
```

Compound (BEGIN...END) Statement

Considerations for Compound Statement
C Examples of Compound Statement

C/COBOL

A compound statement is an embedded SQL statement that groups other embedded SQL statements together.

A compound statement is an SQL/MX extension that you use only in embedded SQL programs in C or COBOL.

```
BEGIN
   SQL-statement;[SQL-statement;]...
END;
```

SQL-statement;[*SQL-statement*;]...

is the SQL statement list between the BEGIN and END keywords. The SQL statements inside a compound statement are executed in sequential order. Therefore, the result of executing a compound statement is exactly the same result as executing the contained statements one at a time in sequential order.

The SQL statements inside a compound statement are atomic. Therefore, if the execution of any statement within the BEGIN and END keywords encounters an error, NonStop SQL/MX automatically rolls back all of the statements.

Considerations for Compound Statement

SQL Statements in the List

You can use most SQL statements inside a compound statement; however, you cannot use transaction statements (BEGIN WORK, COMMIT WORK, ROLLBACK WORK, and SET TRANSACTION), UPDATE STATISTICS, and CONTROL statements.

You can use SELECT INTO to retrieve only one row, but cursors are not allowed in compound statements. You can also use rowsets within compound statements to retrieve multiple rows from database tables.

Executing Compound Statements in a DAM Process

To improve performance, use the CONTROL QUERY DEFAULT OPTS_PUSH_DOWN_DAM option to force NonStop SQL/MX to consider executing compound statements in a NonStop Data Access Manager (DAM) process. Some compound statements, however, should not be executed in a DAM process because they can cause inconsistent data or return the wrong results. For more information about using this option, see the *SQL/MX Query Guide*.

SELECT Statements Within Compound Statements

Every SELECT statement within a BEGIN…END statement should return at least one row. If a SELECT statement within a BEGIN..END statement does not return at least one row, further execution of the compound statement stops and NonStop SQL/MX issues a warning or an error. A warning is displayed if no updates occurred before the SELECT statement that did not return a row. In this case NonStop SQL/MX does not roll back the transaction. An error is displayed if updates occurred before the SELECT statement that did not return a row. Since updates occurred as part of this compound statement NonStop SQL/MX rolls back the transaction. In both the cases the behavior is atomic because none of the statements are executed.

C Examples of Compound Statement

- These INSERT and SELECT statements inside the BEGIN and END keywords execute sequentially:

```
...
EXEC SQL WHENEVER SQLERROR GOTO end_compound;

EXEC SQL
  BEGIN
    INSERT INTO SALES.ORDERS
      (ORDERNUM, ORDER_DATE, DELIV_DATE, SALESREP, CUSTNUM)
       VALUES (:hv_ordernum, :hv_orderdate, :hv_delivdate,
               :hv_salesrep, :hv_custnum);

    SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
      INTO :hv_custnum, :hv_custname,
           :hv_street, :hv_city, :hv_state, :hv_postcode
      FROM SALES.CUSTOMER
      WHERE CUSTNUM = :hv_custnum;
  END;

end_compound:
  ... /* Process the error */
```

DEALLOCATE DESCRIPTOR Statement

C Examples of DEALLOCATE DESCRIPTOR
COBOL Examples of DEALLOCATE DESCRIPTOR

C/COBOL The DEALLOCATE DESCRIPTOR statement deallocates an SQL descriptor area used for storing information necessary for the execution of dynamic SQL statements. The descriptor area was previously allocated with the ALLOCATE DESCRIPTOR statement.

Use DEALLOCATE DESCRIPTOR only in embedded SQL programs in C or COBOL.

```
DEALLOCATE DESCRIPTOR descriptor-name

descriptor-name is:
   [GLOBAL | LOCAL] value-specification
```

descriptor-name

is a *value-specification*—a character literal or host variable with character data type. When DEALLOCATE DESCRIPTOR executes, the content of the host variable (if used) gives the name of the descriptor area.

GLOBAL | LOCAL

specifies the scope of the allocated descriptor area. The default setting is LOCAL. A GLOBAL descriptor area is available to the SQL session. A LOCAL descriptor area is available only to the module or compilation unit in which it was allocated.

An SQL descriptor area must be currently allocated whose name is the value of *descriptor-name* and whose scope is the same scope as specified in the DEALLOCATE DESCRIPTOR statement for the area.

C Examples of DEALLOCATE DESCRIPTOR

- This example uses an SQL string literal as the descriptor name:

```
EXEC SQL DEALLOCATE DESCRIPTOR 'in_sqlda';
```

- This example uses a host variable as the descriptor name:

```
...
EXEC SQL BEGIN DECLARE SECTION;
CHAR desc_name[20];
LONG desc_max;
...
EXEC SQL END DECLARE SECTION;
...
strcpy(desc_name, "in_sqlda");
...
desc_max = 10;
EXEC SQL ALLOCATE DESCRIPTOR :desc_name WITH MAX :desc_max;
...
EXEC SQL DEALLOCATE DESCRIPTOR :desc_name;
```

COBOL Examples of DEALLOCATE DESCRIPTOR

- This example uses an SQL string literal as the descriptor name:

```
EXEC SQL DEALLOCATE DESCRIPTOR 'in_sqlda' END-EXEC.
```

- This example uses a host variable as the descriptor name:

```
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 desc-name   PIC X(20).
01 desc-max    PIC S9(9) comp.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE "in_sqlda" TO desc-name.
...
MOVE 10 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR :desc-name
         WITH MAX :desc-max END-EXEC.
...
EXEC SQL DEALLOCATE DESCRIPTOR :desc-name END-EXEC.
```

DEALLOCATE PREPARE Statement

Considerations for DEALLOCATE PREPARE
C Examples of DEALLOCATE PREPARE
COBOL Examples of DEALLOCATE PREPARE

C/COBOL

The DEALLOCATE PREPARE statement deallocates a prepared SQL statement in a host program. It releases resources held by the prepared statement and allows you to reuse the name of the statement.

Use DEALLOCATE PREPARE only in embedded SQL programs in C or COBOL.

```
DEALLOCATE PREPARE SQL-statement-name

SQL-statement-name is:
   statement-name | ext-statement-name

ext-statement-name is:
   [GLOBAL | LOCAL] value-specification
```

statement-name

is an SQL identifier—the name of a prepared statement to deallocate. The module that contains the DEALLOCATE PREPARE statement must also contain a PREPARE statement for *statement-name*. See Identifiers on page 6-56.

ext-statement-name

is a *value-specification*—a host variable with character data type. When DEALLOCATE PREPARE executes, the content of the *value-specification* must identify a statement previously prepared within the scope of DEALLOCATE PREPARE. The prepared statement must be a cursor specification.

GLOBAL | LOCAL

specifies the scope of the prepared statement. The default setting is LOCAL. A GLOBAL prepared statement can be executed within the SQL session. A LOCAL prepared statement can be executed only within the module or compilation unit in which it was prepared.

A prepared SQL statement must be currently available whose name is the value of *ext-statement-name* and whose scope is the same scope as specified in the DEALLOCATE PREPARE statement.

Considerations for DEALLOCATE PREPARE

Cursor Specification

When you deallocate a prepared statement, any cursor associated with that statement is canceled.

C Examples of DEALLOCATE PREPARE

- Prepare, execute, and deallocate an UPDATE statement with dynamic input parameters:

```
...
strcpy(stmt_buffer,"UPDATE SALES.CUSTOMER"
         " SET CREDIT = ?"
         " WHERE CUSTNUM = CAST(? AS NUMERIC(4) UNSIGNED)")
...
EXEC SQL PREPARE upd_cust FROM :stmt_buffer;
...
/* Input values for parameters into host variables */
scanf("%s",in_credit);
...
scanf("%ld",&in_custnum);
...
EXEC SQL EXECUTE upd_cust USING :in_credit, :in_custnum;
...
EXEC SQL DEALLOCATE PREPARE upd_cust;
```

- This example uses extended statement names:

```
...
strcpy(stmt,"ins_cust1");
EXEC SQL PREPARE :stmt FROM :stmt_buffer;
EXEC SQL EXECUTE :stmt;
EXEC SQL DEALLOCATE PREPARE :stmt;
...
strcpy(stmt,"ins_cust2");
EXEC SQL PREPARE :stmt FROM :stmt_buffer;
EXEC SQL EXECUTE :stmt;
EXEC SQL DEALLOCATE PREPARE :stmt;
...
```

COBOL Examples of DEALLOCATE PREPARE

- Prepare, execute, and deallocate an UPDATE statement with dynamic input parameters:

```
     ...
     MOVE "UPDATE SALES.CUSTOMER SET CREDIT = ?"
        & " WHERE CUSTNUM = CAST(? AS NUMERIC(4) UNSIGNED)"
        & x"00" TO stmt-buffer.
     ...
     EXEC SQL PREPARE upd_cust FROM :stmt-buffer END-EXEC.
     ...
* Input values for parameters into host variables
     ACCEPT in-credit.
     ...
     ACCEPT in-custnum.
     ...
     EXEC SQL EXECUTE upd_cust
                 USING :in-credit, :in-custnum
     END-EXEC.
     ...
     EXEC SQL DEALLOCATE PREPARE upd_cust END-EXEC.
```

- This example uses extended statement names:

```
...
MOVE "ins_cust1" TO stmt.
EXEC SQL PREPARE :stmt FROM :stmt-buffer END-EXEC.
EXEC SQL EXECUTE :stmt END-EXEC.
EXEC SQL DEALLOCATE PREPARE :stmt END-EXEC.
...
MOVE "ins_cust2" TO stmt.
EXEC SQL PREPARE :stmt FROM :stmt-buffer END-EXEC.
EXEC SQL EXECUTE :stmt END-EXEC.
EXEC SQL DEALLOCATE PREPARE :stmt END-EXEC.
...
```

DECLARE CATALOG Declaration

The DECLARE CATALOG declaration is a compiler directive that sets the default catalog for unqualified schema names in static SQL statements that follow the declaration within a compilation unit. The DECLARE SCHEMA declaration sets the default schema name. See DECLARE SCHEMA Declaration on page 3-33.

DECLARE CATALOG is an SQL/MX extension that you use only in embedded SQL programs.

```
DECLARE CATALOG default-catalog
```

default-catalog

is a character string literal that specifies a catalog name. A string literal is enclosed in single quotation marks. `'mycatalog'` is the form, where `mycatalog` is the name you choose.

Considerations for DECLARE CATALOG

Scope of DECLARE CATALOG

You can specify more than one DECLARE CATALOG directive in an embedded SQL program. Each directive replaces the preceding directive and stays in effect until it is replaced by another directive or until the end of the program's compilation unit is reached.

If no DECLARE CATALOG directive is in effect when the SQL/MX compiler encounters an unqualified schema name, the compiler uses the catalog as determined by NonStop SQL/MX. For more information, see Object Naming on page 10-60 and the *SQL/MX Programming Manual for C and COBOL*.

C Examples of DECLARE CATALOG

- Set the default catalog:

```
EXEC SQL DECLARE CATALOG 'SAMDBCAT';
```

COBOL Examples of DECLARE CATALOG

- Set the default catalog:

```
EXEC SQL DECLARE CATALOG 'SAMDBCAT' END-EXEC.
```

DECLARE CURSOR Declaration

C/COBOL

The DECLARE CURSOR declaration or statement specifies a cursor in a host program. It associates the name of the cursor with a query expression that specifies the rows to be retrieved. The program uses the cursor to fetch rows from the result table of the query expression one row at a time.

There are two forms of DECLARE CURSOR—static and dynamic. A static cursor is associated with an actual query expression—for example, a SELECT statement—and a dynamic cursor is associated with a statement name. The static form of DECLARE CURSOR is a declaration, and the dynamic form is an executable statement.

Use DECLARE CURSOR only in embedded SQL programs in C or COBOL.

Pub/Sub

```
DECLARE {cursor-name | ext-cursor-name}
   CURSOR [WITH HOLD | WITHOUT HOLD]
   FOR {cursor-specification
       |ext-statement-name
       |rowset-clause }

cursor-specification is:
   query-expression [order-by-clause] [updatability-clause]

order-by-clause is:
   ORDER BY colname [ASC[ENDING] | DESC[ENDING]]
      [,colname [ASC[ENDING] | DESC[ENDING]]]...

updatability-clause is:
   FOR {READ ONLY | UPDATE [OF colname [,colname]...]}

ext-cursor-name is:
   [GLOBAL | LOCAL] value-specification

ext-statement-name is:
   value-specification

rowset-clause is:
   ROWSET FOR [ INPUT SIZE rowset-size-in]
              [ KEY BY index-identifier]
              [ INPUT SIZE rowset-size-in,
                   KEY BY index-identifier]
              sql-statement SQL terminator
```

See SELECT Statement on page 2-330 for the syntax of *query-expression*.

cursor-name

is an SQL identifier—the name of the cursor being declared. The name is unique within the containing module or compilation unit. The maximum length of a cursor name is 128 characters. See Identifiers on page 6-56.

Pub/Sub `WITH HOLD | WITHOUT HOLD`

specifies whether (WITH) or not (WITHOUT) an application keeps cursors open across transaction boundaries. The default is WITHOUT HOLD. You can use the WITH HOLD clause only with Publish/Subscribe. ■

cursor-specification

is a query expression, an SQL identifier that names a prepared query expression, or in the case of a dynamic cursor, a host variable containing a query expression. It is optionally followed by an ORDER BY clause and a FOR READ ONLY or FOR UPDATE OF clause. See Identifiers on page 6-56.

```
ORDER BY colname [ASC[ENDING] | DESC[ENDING]]
   [,colname [ASC[ENDING] | DESC[ENDING]]]...]
```

specifies the order in which the rows of the query result are presented to the application program. The column name must be the name of a column that is in the select list of *query-expression*.

If the select list in the cursor specification includes an expression (it is not a column name), you must use the AS clause to give a name to the expression. For detailed information, see SELECT Statement on page 2-330.

```
FOR {READ ONLY | UPDATE [OF colname [,colname]...]}
```

specifies whether the cursor is FOR READ ONLY (read-only cursors) or FOR UPDATE OF (updatable cursors). If no column list is specified for an updatable cursor, the column list includes every column of the result table generated from the query expression.

Database modifications—both UPDATE and DELETE operations—are not allowed through read-only cursors. If a column list is specified for an updatable cursor, the columns named in an UPDATE operation must be included in the column list.

ext-cursor-name

is a value specification—a character literal or host variable with character data type. When DECLARE CURSOR executes, the content of the host variable (if used) gives the name of the cursor.

`GLOBAL | LOCAL`

specifies the scope of the value specification for an *`ext-cursor-name`*. The default is LOCAL. The scope of a GLOBAL cursor is the SQL session. The scope of a LOCAL cursor is the module or compilation unit in which DECLARE CURSOR appears.

`ext-statement-name`

is a value specification—a character literal or host variable with character data type.

The *`ext-cursor-name`* and the *`ext-statement-name`* must both be named in the same way—either both as character literals or both as host variables. When host variables are used, the DECLARE CURSOR statement is functionally equivalent to the ALLOCATE CURSOR statement.

`rowset-size-in`

restricts the size of the input rowset to the specified size, which must be less than or equal to the allocated size for the rowset. The size is an integer literal (exact numeric literal) or a host variable whose type is either unsigned short, signed short, unsigned long, or signed long in C and their corresponding equivalents in COBOL. By default, if the size is not specified, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section of the embedded SQL program.

`index-identifier`

is a zero-based index that identifies each row in the matching columns of a SELECT or FETCH statement with the particular search-condition in the WHERE clause that caused the row to be part of the result set. For example, if the row-id value for a certain row in the matching columns is 0 (zero), this row matches the search-condition in the first element of the host variable arrays (array index 0 in C, array index 1 in COBOL) in the WHERE clause.

`SQL-statement`

is any embedded DML statement that uses rowsets directly.

Considerations for DECLARE CURSOR

When DECLARE CURSOR executes, the content of the host variable (if used) must identify a statement previously prepared within the scope of DECLARE CURSOR. The prepared statement must be a cursor specification.

Default for Updatability

You can use updatable cursors only if the query expression involves a single table and simple scan and does not include joins, unions, aggregates, and so on. Suppose that the query expression meets these criteria for updatability:

- If the READONLY_CURSOR attribute is set to TRUE (the default setting), you must declare cursors with the FOR UPDATE clause for the named columns or all columns to be updatable. (This READONLY_CURSOR setting improves cursor performance.)
- If the READONLY_CURSOR attribute is set to FALSE and you omit the FOR READ ONLY clause, all columns except primary key columns are automatically updatable; that is, you do not need to specify the FOR UPDATE clause.
- SQL/MX does not lock the row in the EXCLUSIVE or SHARE mode during a FETCH operation. However, it locks the rows while executing the UPDATE or DELETE statement. Therefore, to prevent the DELETE or UPDATE or DROP statement from parallel processes, specify exclusive lock mode for the SELECT statement while declaring the updatable cursor.

  For information on locking modes, see Considerations for SELECT on page 2-346.

For more information on the READONLY_CURSOR attribute, see Row Maintenance on page 10-75.

Order of Cursor Operations

In static SQL, a cursor declaration must compile before other statements that reference the cursor. In dynamic SQL, a cursor declaration must execute before other statements that reference the cursor.

Declaring Host Variables

The host variables occurring in the cursor specification must be declared within the scope of the associated OPEN statement. Otherwise, an error occurs during preprocessing.

Pub/Sub

WITH HOLD

You can use holdable cursors only for SELECT statements that use the Publish/Subscribe stream access mode or an embedded UPDATE or embedded DELETE. ■

C Examples of DECLARE CURSOR

- This SQL statement defines a static read-only cursor:

```
EXEC SQL DECLARE cursor1 CURSOR FOR
     SELECT COL1, COL2, COL3, COL4 FROM SALES.PARTS
     WHERE  COL2  >= :hostvar2
     READ UNCOMMITTED ACCESS
     ORDER BY COL2;
```

- This SQL statement defines a static updatable cursor. The FOR UPDATE clause lists the columns to be updated:

```
EXEC SQL DECLARE cursor1 CURSOR FOR
     SELECT COL1, COL2, COL3, COL4 FROM SALES.PARTS
     WHERE COL2 >= :hostvar2
     READ COMMITTED ACCESS
     FOR UPDATE OF COL2, COL3, COL4;
```

- This SQL statement defines a dynamic updatable cursor:

```
EXEC SQL BEGIN DECLARE SECTION;
  CHAR query[50];
  ...
EXEC SQL END DECLARE SECTION;

...
strcpy(query,"SELECT COL1, COL2, COL3, COL4"
             " FROM SALES.PARTS");
...
EXEC SQL PREPARE curspec FROM :query;
...
EXEC SQL DECLARE getparts CURSOR FOR curspec;
```

COBOL Examples of DECLARE CURSOR

- This SQL statement defines a static read-only cursor:

```
EXEC SQL DECLARE cursor1 CURSOR FOR
     SELECT COL1, COL2, COL3, COL4 FROM SALES.PARTS
     WHERE  COL2  >= :hostvar2
     READ UNCOMMITTED ACCESS
     ORDER BY COL2
END-EXEC.
```

- This SQL statement defines a static updatable cursor. The FOR UPDATE clause lists the columns to be updated:

```
EXEC SQL DECLARE cursor1 CURSOR FOR
     SELECT COL1, COL2, COL3, COL4 FROM SALES.PARTS
     WHERE COL2 >= :hostvar2
     READ COMMITTED ACCESS
     FOR UPDATE OF COL2, COL3, COL4
END-EXEC.
```

- This SQL statement defines a dynamic updatable cursor:

```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 query  pic x(50).
   ...
   EXEC SQL END DECLARE SECTION END-EXEC.
   ...
   MOVE "SELECT COL1, COL2, COL3, COL4 FROM SALES.PARTS"
     TO query.
   ...
   EXEC SQL PREPARE curspec FROM :query END-EXEC.
```

```
...
EXEC SQL DECLARE getparts CURSOR FOR curspec END-EXEC.
```

Publish/Subscribe Examples of DECLARE CURSOR

Suppose that these SQL/MP tables and index (and the metadata mappings) have been created:

```
CREATE TABLE $db.dbtab.tab1 (a INT, b INT, c INT);
CREATE TABLE $db.dbtab.tab2 (a INT, b INT, c INT);
CREATE INDEX $db.dbtab.itab1 ON tab1(b, c);
```

```
CREATE SQLMP ALIAS cat.sch.tab1 $db.dbtab.tab1;
CREATE SQLMP ALIAS cat.sch.tab2 $db.dbtab.tab2;
```

- This example shows a holdable cursor:

```
DECLARE SCHEMA cat.sch;

EXEC SQL
  DECLARE holdable_cursor CURSOR WITH HOLD FOR
    SELECT * FROM (DELETE FROM STREAM(tab1)) tab1;

EXEC SQL BEGIN WORK;
EXEC SQL OPEN holdable_cursor;
for(;;) {
        EXEC SQL FETCH holdable_cursor INTO :hv;
        EXEC SQL COMMIT;
        EXEC SQL BEGIN WORK;
}
```

DECLARE MPLOC Declaration

The DECLARE MPLOC declaration is a compiler directive that sets the default volume and subvolume for unqualified Guardian physical object names in static SQL statements that follow the declaration within a compilation unit.

C/COBOL DECLARE MPLOC is used by the preprocessor when the INVOKE directive is not fully qualified. ■ It is also used in embedded programs to access SQL/MP tables in static SQL statements. Otherwise, you must use CREATE SQLMP ALIAS to map an SQL/MP table to an ANSI name.

You must precede DECLARE MPLOC statements with the DECLARE NAMETYPE 'NSK' statement. Otherwise, the program defaults to ANSI type and DECLARE MPLOC is ignored.

DECLARE MPLOC is an SQL/MX extension that you use only in embedded SQL programs.

```
DECLARE MPLOC default-mploc
```

default-mploc

is a character string literal that specifies the Guardian physical name of a subvolume. A string literal is enclosed in single quotation marks.

The form is: `'[\node.]$volume.subvolume'`

If you do not specify `\node`, the default is the Guardian system named in your =_DEFAULTS define. By using the `\node` name, you can have multiple database access over the network during processing.

Considerations for DECLARE MPLOC

C/COBOL

Preprocessor and INVOKE Directive

The way you specify DECLARE MPLOC affects whether the preprocessor preserves or overrides the INVOKE directive:

DECLARE MPLOC is specified as...	INVOKE Directive	Preprocessor Action
`\`*`node`*`.$`*`vol`*`.`*`subvol`*`.`*`filename`*	*`table`* is not fully qualified and does not contain `\`*`node`*.	Overrides the INVOKE directive with DECLARE MPLOC names.
`\`*`node`*`.$`*`vol`*`.`*`subvol`*`.`*`filename`*	*`table`* contains `\`*`node`*.	Preserves the INVOKE directive and does not use DECLARE MPLOC names.
`$`*`vol`*`.`*`subvol`*`.`*`filename`*	*`table`* is not fully qualified and does not contain `\`*`node`*.	Overrides the INVOKE directive with DECLARE MPLOC names.
`$`*`vol`*`.`*`subvol`*`.`*`filename`*	*`table`* contains `\`*`node`*.	Preserves the INVOKE directive and does not use DECLARE MPLOC names.

■

Scope of DECLARE MPLOC

You can specify more than one DECLARE MPLOC directive in an embedded SQL program. Each directive replaces the preceding directive and stays in effect until it is replaced by another directive or until the end of the program's compilation unit is reached.

If no DECLARE MPLOC directive is in effect when the SQL/MX compiler encounters an unqualified Guardian physical name, the compiler uses the volume and subvolume as determined by NonStop SQL/MX. For more information, see Object Naming on page 10-60 and the *SQL/MX Programming Manual for C and COBOL*.

C Examples of DECLARE MPLOC

- Set the default volume and subvolume:

```
EXEC SQL DECLARE MPLOC '$MYVOL.MYSUBVOL';
```

COBOL Examples of DECLARE MPLOC

- Set the default volume and subvolume:

```
EXEC SQL DECLARE MPLOC '$MYVOL.MYSUBVOL' END-EXEC.
```

DECLARE NAMETYPE Declaration

The DECLARE NAMETYPE declaration is a compiler directive that sets the default NAMETYPE attribute value to ANSI or NSK for static SQL statements that follow the declaration within a compilation unit.

C/COBOL DECLARE NAMETYPE is used by the preprocessor when the INVOKE directive is not fully qualified. ■

DECLARE NAMETYPE is an SQL/MX extension that you use only in embedded SQL programs.

```
DECLARE NAMETYPE default-nametype
```

default-nametype

is a character string literal that specifies the NAMETYPE attribute value used to refer to SQL/MP database objects. `'ANSI'` indicates logical names (ANSI), and `'NSK'` indicates physical Guardian names.

If you do not specify DECLARE NAMETYPE, the default NAMETYPE is ANSI.

Considerations for DECLARE NAMETYPE

Scope of DECLARE NAMETYPE

You can specify more than one DECLARE NAMETYPE directive in an embedded SQL program. Each directive replaces the preceding directive and stays in effect until it is replaced by another directive or until the end of the program's compilation unit is reached.

If no DECLARE NAMETYPE directive is in effect when the SQL/MX compiler encounters an unqualified object name, the compiler uses the value of the NAMETYPE attribute as determined by NonStop SQL/MX. For more information, see Object Naming on page 10-60 and the *SQL/MX Programming Manual for C and COBOL*.

C Examples of DECLARE NAMETYPE

- Set the default NAMETYPE attribute value to use Guardian physical names:

```
EXEC SQL DECLARE NAMETYPE 'NSK';
```

COBOL Examples of DECLARE NAMETYPE

- Set the default NAMETYPE attribute value to use Guardian physical names:

```
EXEC SQL DECLARE NAMETYPE 'NSK' END-EXEC.
```

DECLARE SCHEMA Declaration

The DECLARE SCHEMA declaration is a compiler directive that sets the default schema (and optionally, the catalog) for unqualified object names in static SQL statements that follow the declaration within a compilation unit. The DECLARE CATALOG declaration also sets the default catalog. See DECLARE CATALOG Declaration on page 3-21.

DECLARE SCHEMA is an SQL/MX extension that you use only in embedded SQL programs.

```
DECLARE SCHEMA default-schema
```

default-schema

is a character string literal enclosed in single quotation marks (') that specifies the default schema (and optionally the catalog). Examples are `'sales'` for only a default schema or `'samdbcat.sales'` for both a default schema and catalog.

Considerations for DECLARE SCHEMA

Scope of DECLARE SCHEMA

You can specify more than one directive in an embedded SQL program. Each directive replaces the preceding directive and stays in effect until it is replaced by another directive or until the end of the program's compilation unit is reached.

If no DECLARE SCHEMA declaration is in effect when the SQL/MX compiler encounters an unqualified object name, the compiler uses the SCHEMA attribute as determined by NonStop SQL/MX. For more information, see Object Naming on page 10-60 and the *SQL/MX Programming Manual for C and COBOL*.

C Examples of DECLARE SCHEMA

- Set the default catalog and schema:

```
EXEC SQL DECLARE CATALOG 'SAMDBCAT';
EXEC SQL DECLARE SCHEMA 'SALES';
```

- Set the default catalog and schema within one statement:

```
EXEC SQL DECLARE SCHEMA 'SAMDBCAT.SALES';
```

COBOL Examples of DECLARE SCHEMA

- Set the default catalog and schema:

```
EXEC SQL DECLARE CATALOG 'SAMDBCAT' END-EXEC.
EXEC SQL DECLARE SCHEMA 'SALES' END-EXEC.
```

DESCRIBE Statement

C Examples of DESCRIBE
COBOL Examples of DESCRIBE

C/COBOL

The DESCRIBE statement obtains information, including data types of columns, about dynamic input and output parameters contained in a prepared statement. A parameter is a placeholder for a value to be supplied when the statement executes.

There are two forms of the DESCRIBE statement:

- DESCRIBE INPUT—initializes the input SQL descriptor area based on the input parameters for a prepared statement
- DESCRIBE [OUTPUT]—stores descriptions into the SQL descriptor area of output parameters (usually SELECT columns) from a prepared statement

Use DESCRIBE only in embedded SQL programs in C or COBOL.

```
DESCRIBE {INPUT | [OUTPUT]} SQL-stmt-name using-descriptor

SQL-stmt-name is:
   statement-name | ext-statement-name

ext-statement-name is:
   [GLOBAL | LOCAL] value-specification

using-descriptor is:
   USING SQL DESCRIPTOR descriptor-name

descriptor-name is:
   [GLOBAL | LOCAL] value-specification
```

statement-name

is an SQL identifier—the name of a prepared statement. The module that contains DESCRIBE must also contain a PREPARE statement for *statement-name*. See Identifiers on page 6-56.

ext-statement-name

is a *value-specification*—a host variable with character data type. When DESCRIBE executes, the content of the value specification must identify a statement previously prepared within the scope of DESCRIBE.

`GLOBAL | LOCAL`

specifies the scope of the prepared statement. The default is LOCAL. A GLOBAL prepared statement can be described and executed within the SQL session. A LOCAL prepared statement can be described and executed only within the module or compilation unit in which it was prepared.

A prepared SQL statement must be currently available whose name is the value of *`ext-statement-name`* and whose scope is the same scope as specified in the DESCRIBE INPUT statement.

`USING SQL DESCRIPTOR` *`descriptor-name`*

identifies the SQL descriptor area for the parameters of *`SQL-statement-name`*. An SQL descriptor area must be currently allocated whose name is the value of *`descriptor-name`* and whose scope is the same scope as specified in the DESCRIBE statement.

When DESCRIBE INPUT executes, NonStop SQL/MX stores information for each input parameter of the prepared statement. Each parameter has an item descriptor.

When DESCRIBE OUTPUT executes, NonStop SQL/MX stores information about each column specified in the select list for the prepared statement. Each column has an item descriptor.

`descriptor-name`

is a *`value-specification`*—a character literal or host variable with a character data type. When DESCRIBE executes, the content of the host variable (if used) gives the name of the descriptor area.

C Examples of DESCRIBE

- Returns descriptions of input parameters for the prepared statement identified by `:stmt_name` to an SQL descriptor area identified by the host variable `:input_sqlda:`

```
EXEC SQL DESCRIBE INPUT :stmt_name
     USING SQL DESCRIPTOR :input_sqlda;
```

- Returns descriptions of output variables specified in the prepared statement identified by S1 to the SQL descriptor area identified by the character literal `'output_sqlda'`:

```
EXEC SQL DESCRIBE OUTPUT S1
     USING SQL DESCRIPTOR 'output_sqlda';
```

- Prepare a statement, allocate input and output descriptor areas, and describe the input and output descriptor areas:

```
...
strcpy(stmt_buffer,"SELECT * FROM EMPLOYEE"
       " WHERE EMPNUM = CAST(? AS NUMERIC(4) UNSIGNED)");
```

```
...
EXEC SQL PREPARE S1 FROM :stmt_buffer;
...
desc_max = 1;
EXEC SQL ALLOCATE DESCRIPTOR 'in_args' WITH MAX :desc_max;
desc_max = 6;
EXEC SQL ALLOCATE DESCRIPTOR 'out_cols' WITH MAX :desc_max;
...
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR 'in_args';
EXEC SQL DESCRIBE OUTPUT S1 USING SQL DESCRIPTOR 'out_cols';
...
```

COBOL Examples of DESCRIBE

- Return descriptions of input parameters for the prepared statement identified by `:stmt-name` to the SQL descriptor area identified by the host variable `:input_sqlda`:

```
EXEC SQL DESCRIBE INPUT :stmt-name
     USING SQL DESCRIPTOR :input_sqlda
END-EXEC.
```

- Return descriptions of output variables specified in the prepared statement identified by S1 to the SQL descriptor area identified by the character literal `'output_sqlda'`:

```
EXEC SQL DESCRIBE OUTPUT S1
     USING SQL DESCRIPTOR 'output_sqlda'
END-EXEC.
```

- Prepare a statement and allocate and describe the input and output descriptor areas:

```
...
MOVE "SELECT * FROM EMPLOYEE"
   & " WHERE EMPNUM = CAST(? AS NUMERIC(4) UNSIGNED)"
TO stmt-buffer.
...
EXEC SQL PREPARE S1 FROM :stmt-buffer END-EXEC.
...
MOVE 1 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR 'in_args'
           WITH MAX :desc-max END-EXEC.
MOVE 6 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR 'out_cols'
           WITH MAX :desc-max END-EXEC.
...
EXEC SQL DESCRIBE INPUT S1
         USING SQL DESCRIPTOR 'in_args'
END-EXEC.
EXEC SQL DESCRIBE OUTPUT S1
         USING SQL DESCRIPTOR 'out_cols'
END-EXEC.
...
```

END DECLARE SECTION Declaration

C/COBOL

END DECLARE SECTION is a preprocessor directive that ends SQL declarations in a host program. SQL declarations are used to define host variables to be used in SQL/MX statements—for example, to transfer data to and from a database.

Use END DECLARE SECTION only in embedded SQL programs in C or COBOL.

```
END DECLARE SECTION
```

See BEGIN DECLARE SECTION Declaration on page 3-9.

C Examples of END DECLARE SECTION

- This example shows a declaration section in a C program:

```
EXEC SQL BEGIN DECLARE SECTION;
  SHORT length;
  CHAR  data[10];
EXEC SQL END DECLARE SECTION;
```

C++ Examples of END DECLARE SECTION

- This example shows a declaration section within a class in a C++ program. Member functions using these host variables must be defined within the visible scope of the class.

```
class jobsql {
// Class member host variables
EXEC SQL BEGIN DECLARE SECTION;
  short length;
  VARCHAR data[19];
EXEC SQL END DECLARE SECTION;
public:
...  // Member functions referencing these host variables
  }
};  // End of jobsql class definition
```

COBOL Examples of END DECLARE SECTION

- This example shows a declaration section in a COBOL program:

```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 length   pic 9(4)comp.
01 data     pic x(10).
   EXEC SQL END DECLARE SECTION END-EXEC.
```

EXEC SQL Directive

C/COBOL EXEC SQL is a preprocessor directive that begins an embedded SQL declaration or statement.

Use EXEC SQL only in embedded SQL programs in C or COBOL.

```
EXEC SQL {sql-declaration | sql-statement} sql-terminator
```

sql-declaration

is any embedded SQL declaration.

sql-statement

is any embedded SQL statement.

sql-terminator

terminates the SQL declaration or statement. For a C program, semicolon (;) is the terminator. For a COBOL program, END-EXEC is the terminator.

Considerations for EXEC SQL

Using Host Language Comments

You can use host language comments within SQL statements:

- C comments have the form: `/* ... */`. The comment is not restricted to one line.
- COBOL comments have the form: `* ...` The asterisk (*) is in the first column of the source code line in free format and in the seventh column of the source code line for fixed format. The comment is restricted to one line.

Examples of EXEC SQL

For examples of the EXEC SQL directive, see the various C and COBOL examples throughout this section.

EXECUTE IMMEDIATE Statement

C/COBOL

The EXECUTE IMMEDIATE statement compiles and executes an SQL statement whose text is contained in a host variable. The SQL statement supplied cannot have any input or output parameters and must be a preparable statement.

Use EXECUTE IMMEDIATE only in embedded SQL programs in C or COBOL.

```
EXECUTE IMMEDIATE SQL-statement-variable
```

SQL-statement-variable

is a *value-specification*—a host variable with character data type. When EXECUTE IMMEDIATE executes, the content of the value specification give the text of the SQL statement to be compiled and executed. The SQL statement cannot contain parameters or refer to host variables.

Considerations for EXECUTE IMMEDIATE

Parameters

If the statement to be compiled and executed contains input or output parameters, you must use separate PREPARE and EXECUTE statements.

C Examples of EXECUTE IMMEDIATE

- Execute an SQL statement whose text is contained in the host variable named `:statement`:

```
EXEC SQL EXECUTE IMMEDIATE :statement;
```

COBOL Examples of EXECUTE IMMEDIATE

- Execute an SQL statement whose text is contained in the host variable named `:statement`:

```
EXEC SQL EXECUTE IMMEDIATE :statement END-EXEC.
```

FETCH Statement

Considerations for FETCH
C Examples of FETCH
COBOL Examples of FETCH

The FETCH statement is an SQL statement that positions a cursor on the next row of the result table defined by the cursor specification and retrieves values from that row, leaving the cursor positioned at that row.

C/COBOL When you use the FETCH statement with rowset host variables or descriptors with appropriate rowset fields set, it retrieves values from multiple, consecutive rows in the result table. The number of rows from which values are retrieved is the given by the declared length of the rowset and the number of rows in the result table, whichever is smaller. After the values have been retrieved NonStop SQL/MX positions the cursor on the last row that was read.

In dynamic SQL, the cursor name is provided at execution time, and the USING or INTO clause can specify a target list of host variables or an SQL descriptor area for the output values. In static SQL, the INTO clause provides a target list of host variables. Otherwise, there is no difference in the static and dynamic forms of FETCH. ■

FETCH is one of several statements (including COMMIT, ROLLBACK, and SET TRANSACTION) that do not generate a system-defined transaction.

Use FETCH only in embedded SQL programs.

C/COBOL

```
FETCH {cursor-name | ext-cursor-name}
   {USING | INTO} {argument-list | descriptor-spec}

ext-cursor-name is:
   [GLOBAL | LOCAL] value-specification

argument-list is:
   variable-spec [,variable-spec]...

descriptor-spec is:
   SQL DESCRIPTOR descriptor-name

variable-spec is:
   :variable-name [[INDICATOR] :indicator-name] ■
```

C/COBOL *cursor-name*

is an SQL identifier—the name of the cursor being used to fetch a row of values. The cursor must be open. See Identifiers on page 6-56. ■

C/COBOL

ext-cursor-name

is a *value-specification*—a character literal or a host variable with character data type. When FETCH executes, the content of the host variable (if used) gives the name of the open cursor. ■

C/COBOL

GLOBAL | LOCAL

specifies scope. The default is LOCAL. The scope of a GLOBAL cursor is the SQL session. The scope of a LOCAL cursor is the module or compilation unit in which FETCH appears. The containing module must include a DECLARE CURSOR with the same cursor name. ■

C/COBOL

{USING | INTO} {*argument-list* | *descriptor-spec*}

specifies host variables, rowset host variables, or, in the case of a dynamic cursor, an SQL descriptor area in which to return the values in the result row of the cursor specification. For a static cursor, the number of row values must be equal to the number of specified host variables, and the data type of each source value must be compatible with the data type of its target host variable. The first value in the result row is assigned to the first host variable, the second value to the second variable, and so on.

If you use rowset host variables or descriptors with the appropriate rowset fields set, values from multiple, consecutive rows are moved into the rowset host variables with a single execution of the FETCH statement.

In static SQL, you use the INTO keyword. In dynamic SQL, you can use either USING or INTO. The use of the keyword USING is an SQL/MX extension.

:*variable-name* [[INDICATOR] :*indicator-name*]

is a variable specification—a host variable or rowset host variable with, optionally, an indicator variable or rowset indicator variable. A variable name begins with a colon (:).

The data type of an indicator variable is exact numeric with a scale of 0. If the data returned in the host variable is null, the indicator parameter is set to a value less than zero. If character data returned is truncated, the indicator parameter is set to the length of the string in the database. ■

Java

INTO *argument-list*

specifies host variables. The number of row values must be equal to the number of specified host variables, and the data type of each source value must be compatible with the data type of its target host variable. The first value in the result row is assigned to the first host variable, the second value to the second variable, and so on.

`:variable-name`

is a variable specification—a host variable. A variable name begins with a colon (`:`). ■

Considerations for FETCH

Authorization Requirements

FETCH requires read access to any tables or views associated with the cursor or iterator. Updating fetched rows requires write access to the table or view.

Ordering Fetched Rows

Successive executions of FETCH retrieve successive rows in the result table of the cursor specification or iterator.

C/COBOL To control the order in which the rows appear, include an ORDER BY clause in the cursor specification part of DECLARE CURSOR or in the prepared statement in the case of a dynamic cursor. ■

Java To control the order in which the rows appear, include an ORDER BY clause in the SELECT statement that is bound to the iterator. ■

Too Many Values or Too Many Variables

If the number of host variables is different from the number of columns in the result table, the execution of FETCH raises an error condition.

C/COBOL

Using Extended Dynamic Cursors

The name of an extended dynamic cursor is not known until run time. When FETCH executes, the name must identify an open cursor within the same scope. ■

C/COBOL

Status Information

You must declare the variables SQLSTATE or SQLCODE in your module or compilation unit. For more information on declaring SQLCODE and SQLSTATE, see the *SQL/MX Programming Manual for C and COBOL*.

FETCH returns a five-character status code to SQLSTATE, whose values include:

00000	The FETCH was successful.
02000	The result table is empty or the end of the table was encountered.
22xxx	Data exception condition.

For more information on the ANSI SQL:1999 SQLSTATE class and subclass values, see the *SQL/MX Programming Manual for C and COBOL*.

FETCH also returns an integer status code to SQLCODE, as follows:

0	The FETCH was successful.
100	The result table is empty or the end of the table was encountered.
> 0	A warning was issued.
< 0	An error occurred.

SQLSTATE, the SQL:1999 standard, is the preferred status code for NonStop SQL/MX. ■

C Examples of FETCH

- Suppose that you have a cursor that returns information from the PARTS table. The host variables are declared in a declaration section, and the cursor declaration lists the columns to be retrieved. The FETCH statement lists host variables to receive the values returned for each column:

```
/* Variable declarations */
long SQLCODE;
...
/* Host variable declarations */
EXEC SQL BEGIN DECLARE SECTION;
   CHAR SQLSTATE[6];
   ... hostvar;
   ... hostvar1;
   ... hostvar2;
   ... hostvar3;
EXEC SQL END DECLARE SECTION;
...
/* Declare cursor. */
EXEC SQL DECLARE cursor1 CURSOR FOR
         SELECT  COL1, COL2, COL3
         FROM    PARTS
         WHERE   COL1  >= :hostvar
         ORDER BY COL1
         READ UNCOMMITTED ACCESS;
...
/* Open cursor. */
EXEC SQL OPEN cursor1;
...
/* Fetch current row. */
EXEC SQL FETCH cursor1
         INTO :hostvar1, :hostvar2, :hostvar3;
if SQLCODE = 100 goto ... ;
...
/* Close cursor. */
EXEC SQL CLOSE cursor1;
```

- This example uses extended cursor and statement names:

```
scanf("%s", in_curspec);
...
EXEC SQL PREPARE :curspec FROM :in_curspec;
EXEC SQL ALLOCATE :extcur CURSOR FOR :curspec;
...
EXEC SQL OPEN :extcur;
...
desc_max = 10;
ALLOCATE DESCRIPTOR 'fetch_sqlda' WITH MAX :desc_max;
DESCRIBE OUTPUT :curspec USING SQL DESCRIPTOR 'fetch_sqlda';
FETCH :extcur INTO SQL DESCRIPTOR 'fetch_sqlda';
...            /* Process values in SQL descriptor area. */
EXEC SQL CLOSE :extcur;
```

COBOL Examples of FETCH

- Suppose that you have a cursor that returns information from the PARTS table. The host variables are declared in a declaration section, and the cursor declaration lists the columns to be retrieved. The FETCH statement lists host variables to receive the values returned for each column.

```
* Variable declarations
  01 SQLCODE    PIC S9(9) comp.
  ...
* Host variable declarations
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 SQLSTATE   PIC X(5).
  01 hostvar  ... .
  01 hostvar1  ... .
  01 hostvar2  ... .
  01 hostvar3  ... .
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
* Declare cursor.
     EXEC SQL DECLARE cursor1 CURSOR FOR
         SELECT  COL1, COL2, COL3
         FROM    PARTS
         WHERE   COL1  >= :hostvar
         ORDER BY COL1
         READ UNCOMMITTED ACCESS
     END-EXEC.
     ...
* Open cursor.
     EXEC SQL OPEN cursor1 END-EXEC.
     ...
* Fetch current row.
     EXEC SQL FETCH cursor1
         INTO :hostvar1, :hostvar2, :hostvar3
     END-EXEC.
     IF SQLCODE = 100 GOTO nodata.
     ...
```

```
* Close cursor.
      EXEC SQL CLOSE cursor1 END-EXEC.
      ...
  nodata SECTION.
      ...
```

- This example uses extended cursor and statement names:

```
      ACCEPT in-curspec.
      ...
      EXEC SQL PREPARE :curspec FROM :in-curspec END-EXEC.
      EXEC SQL ALLOCATE :extcur CURSOR FOR :curspec END-EXEC.
      ...
      EXEC SQL OPEN :extcur END-EXEC.
      ...
      MOVE 10 TO desc-max.
      EXEC SQL ALLOCATE DESCRIPTOR 'fetch_sqlda'
        WITH MAX desc-max END-EXEC.

      EXEC SQL DESCRIBE OUTPUT :curspec
        USING SQL DESCRIPTOR 'fetch_sqlda' END-EXEC.

      EXEC SQL FETCH :extcur
        INTO SQL DESCRIPTOR 'fetch_sqlda' END-EXEC.
* Process values in SQL descriptor area.
      ...
      EXEC SQL CLOSE :extcur END-EXEC.
```

GET DESCRIPTOR Statement

SQL Item Descriptor Area of GET DESCRIPTOR
SQL Descriptor Area Data Type Declarations of GET DESCRIPTOR
Considerations for GET DESCRIPTOR
C Examples of GET DESCRIPTOR
COBOL Examples of GET DESCRIPTOR

C/COBOL

The GET DESCRIPTOR statement retrieves information from an SQL descriptor area. An application program can either retrieve the count of item descriptors with information or the value of a specific field within a specific item.

Use GET DESCRIPTOR only in embedded SQL programs in C or COBOL.

```
GET DESCRIPTOR descriptor-name get-descriptor-info

descriptor-name is:
   [GLOBAL | LOCAL] value-specification

get-descriptor-info is:
     variable-name = COUNT
   | variable-name = ROWSET_SIZE
   | VALUE item-number get-item-info [,get-item-info]...

get-item-info is:
   variable-name = descriptor-item-name

descriptor-item-name is:
     CHARACTER_SET_NAME
   | CHARACTER_SET_NAME | CHAR SET
   | CHARACTER_SET_CATALOG
   | CHARACTER_SET_SCHEMA
   | COLLATION
   | COLLATION_CATALOG
   | COLLATION_NAME
   | COLLATION_SCHEMA
   | DATETIME_CODE
   | HEADING
   | INDICATOR_DATA | INDICATOR
   | INDICATOR_POINTER
   | INDICATOR_TYPE
   | LEADING_PRECISION
   | LENGTH
   | NAME
   | NULLABLE
   | OCTET_LENGTH
   | PARAMETER_MODE
   | PARAMETER_ORDINAL_POSITION
   | PRECISION
   | RETURNED_LENGTH
   | RETURNED_OCTET_LENGTH
   | ROWSET_IND_LAYOUT_SIZE
   | ROWSET_VAR_LAYOUT_SIZE
```

```
    | SCALE
    | TYPE     | TYPE_FS
    | UNNAMED
    | VARIABLE_DATA | DATA
    | VARIABLE_POINTER
```

descriptor-name

is a *value-specification*—a character literal or host variable with character data type. The named SQL descriptor area must be currently allocated.

variable-name = COUNT

retrieves the count of item descriptors with information and stores the count in the named host variable. COUNT is the number of input dynamic parameters or output dynamic parameters (from a stored procedure or from select list columns) described in the descriptor area.

variable-name = ROWSET_SIZE

retrieves the length of rowset variables specified in this descriptor descriptors and stores the length in the named host variable. ROWSET_SIZE is the common length of all input or output rowsets described in the descriptor area.

VALUE *item-number get-item-info* [,*get-item-info*]...

retrieves the value of a specific field within a specific item. See SQL Item Descriptor Area of GET DESCRIPTOR on page 3-48.

item-number

refers to a particular item in the SQL descriptor area. The data type of the item number must be exact numeric, and its value must be less than the maximum number of occurrences specified when the SQL descriptor area was allocated. If the item number exceeds the value of COUNT, a completion condition is raised (no data). See ALLOCATE DESCRIPTOR Statement on page 3-6.

variable-name = *descriptor-item-name*

specifies the host variable in which to store information and the field from which to retrieve the information. The host variable must be of an appropriate data type and size for the information being retrieved.

SQL Item Descriptor Area of GET DESCRIPTOR

Table 3-1 describes the items in the descriptor area for GET DESCRIPTOR. For character fields with lengths greater than or equal to 128, declare the corresponding host variables as type `VARCHAR` with length 129 (with an extra byte for the null terminator) in C or type `PIC X` with length 128 in COBOL.

Table 3-1. GET DESCRIPTOR Items (page 1 of 4)

Name of Field	Data Type and Description
CHARACTER_SET_NAME*	Character string, minimum length >= 128. One, two, or three-part name of the character set.
CHARACTER_SET_NAME	Character string, minimum length >= 128. One-part character set name.
CHARACTER_SET _CATALOG	Character string, minimum length >= 128. Catalog part of the character set name.
CHARACTER_SET _SCHEMA	Character string, minimum length >= 128. Schema part of the character set name.
COLLATION*	Character string, minimum length >= 128. One, two, or three-part name of the collation.
COLLATION_CATALOG	Character string, minimum length >= 128. Catalog part of the collation name.
COLLATION_NAME	Character string, minimum length >= 128. One-part collation name.
COLLATION_SCHEMA	Character string, minimum length >= 128. Schema part of the collation name.
DATETIME_CODE	Exact numeric, scale 0. Codes for DATETIME type: 1 date; 2 time; 3 timestamp. Codes for INTERVAL subfields: 1 year; 2 month; 3 day; 4 hour; 5 minute; 6 second; 7 year to month; 8 day to hour; 9 day to minute; 10 day to second; 11 hour to minute; 12 hour to second; 13 minute to second. This field is equivalent to the ANSI-named DATETIME_INTERVAL_CODE field. You cannot use the ANSI name for this field.**
HEADING*	Character string, minimum length >= 128. Heading for associated column.

* The statement item is an SQL/MX extension.
** The SQL/MX name is different from the ANSI name.

Table 3-1. GET DESCRIPTOR Items (page 2 of 4)

Name of Field	**Data Type and Description**
INDICATOR_DATA	Exact numeric, scale 0. Value for the indicator variable of VARIABLE_DATA: 0 INDICATOR_DATA is not null. <0 INDICATOR_DATA is null. >0 VARIABLE_DATA was truncated and INDICATOR_DATA is the length of the source data. This field is equivalent to the ANSI-named INDICATOR field. You can also use INDICATOR as the name of the field.**
INDICATOR_POINTER*	Pointer to the value of INDICATOR_DATA.
INDICATOR_TYPE*	Exact numeric, scale 0. Type of INDICATOR_DATA. The default type is `short`. Values for INDICATOR_TYPE are: -1 numeric data is negative 0 (optional) numeric data is positive
LEADING_PRECISION	Exact numeric, scale 0. Precision of interval start field. This field is equivalent to the ANSI-named DATETIME_INTERVAL_PRECISION field. You cannot use the ANSI name for this field.**
LENGTH	Exact numeric, scale 0. Length in characters for strings or in bytes for other data types.
NAME	Character string, minimum length >= 128. Name of the associated column or name of the output parameter of a stored procedure (if specified in the CREATE PROCEDURE statement).
NULLABLE	Exact numeric, scale 0. Whether the associated column is nullable. Codes: 1 nullable; 0 not nullable. For a dynamic parameter, NULLABLE is set to 1, indicating that the dynamic parameter can have a null value.
OCTET_LENGTH	Exact numeric, scale 0. Length in bytes for the field.
PARAMETER_MODE	Smallint. Indicates whether the associated formal parameter of the stored procedure was declared as IN, OUT, or, INOUT. Four possible values: 0 PARAMETER_MODE_UNDEFINED 1 PARAMETER_MODE_IN 2 PARAMETER_MODE_OUT 4 PARAMETER_MODE_INOUT The default value is 0 (zero), indicating that the parameter mode is undefined. For all SQL statements other than the CALL statement, PARAMETER_MODE is undefined.

* The statement item is an SQL/MX extension.
** The SQL/MX name is different from the ANSI name.

Table 3-1. GET DESCRIPTOR Items (page 3 of 4)

Name of Field	Data Type and Description
PARAMETER_ORDINAL_POSITION	Smallint. Indicates the position of a formal parameter in the signature of a stored procedure that corresponds to a described dynamic parameter. Values start at 1. A value of 0 (zero) means the position is undefined. For all SQL statements other than the CALL statement, PARAMETER_ORDINAL_POSITION is undefined.
PRECISION	Exact numeric, scale 0. Precision for numeric types. PRECISION specifies the total number of digits and cannot exceed 18.
RETURNED_LENGTH	Exact numeric, scale 0. Returned length in characters for strings or in bytes for other data types.
RETURNED_OCTET_LENGTH	Exact numeric, scale 0. Returned length in bytes.
ROWSET_IND_LAYOUT_SIZE	Exact numeric, scale 0. Size of an individual array element in a rowset host variable. A value 0 (zero) in this field denotes that the host variable is not of rowset type and is a scalar host variable.
ROWSET_VAR_LAYOUT_SIZE	Exact numeric, scale 0. Size of an individual array element in a rowset host variable. A value 0 (zero) in this field denotes that the host variable is not of rowset type and is a scalar host variable.
SCALE	Exact numeric, scale 0. Scale for exact numeric types. SCALE specifies the number of digits to the right of the decimal point.
TYPE	Exact numeric, scale 0. ANSI codes for data type: 1 CHARACTER; 2 NUMERIC; 3 DECIMAL; 4 INTEGER; 5 SMALLINT; 6 IEEE FLOAT; 7 IEEE REAL; 8 DOUBLE precision; 9 DATE, TIME, or TIMESTAMP; 10 INTERVAL; 12 CHARACTER VARYING. SQL/MX extensions: -101 character uppercase; -201 numeric unsigned; -301 decimal unsigned; -302 decimal large; -303 decimal large unsigned; -401 integer unsigned; -402 largeint; -502 smallint unsigned; -601 character varying with length specified in first two bytes. See Version Differences for TYPE and TYPE_FS on page 3-53.

* The statement item is an SQL/MX extension.
** The SQL/MX name is different from the ANSI name.

Table 3-1. GET DESCRIPTOR Items (page 4 of 4)

Name of Field	Data Type and Description
TYPE_FS*	Exact numeric, scale 0. These codes are SQL/MP-specific codes returned in the SQL descriptor area, as shown under SQL Descriptor Area Data Type Declarations of GET DESCRIPTOR on page 3-51. If you set TYPE_FS, you must also set LENGTH. This field does not provide the ANSI codes for data type; ANSI codes are provided by TYPE. See Version Differences for TYPE and TYPE_FS on page 3-53.
UNNAMED	Exact numeric, scale 0. Whether the associated select list item is a named column. Codes: 1 unnamed; 0 named.
VARIABLE_DATA	Actual data associated with the dynamic parameter. The type, length, name, and so on, are determined by other fields. This field is equivalent to the ANSI-named DATA field. You can also use DATA as the name of the field.**
VARIABLE_POINTER*	Pointer to the value of VARIABLE_DATA.

* The statement item is an SQL/MX extension.
** The SQL/MX name is different from the ANSI name.

Note. You must declare the host variables for the exact numeric fields in the above table as 32-bit integers. However, declare the host variables as long for VARIABLE_POINTER and INDICATOR_POINTER.

SQL Descriptor Area Data Type Declarations of GET DESCRIPTOR

The TYPE_FS field can have the values shown in Table 3-2. Use the declarations in the `sqlci.h` header file for the TYPE_FS field in a C or C++ program. The `sqlci.h` header file is automatically included in embedded SQL source files. Therefore, you can use these declarations without adding `#include` directives.

Table 3-2. Descriptor Area Data Type Declarations (page 1 of 2)

Value	Declaration in `sqlci.h` File	Description
Character Data Types (0 – 127)		
0	_SQLDT_ASCII_F	Fixed-length single-byte character
1	_SQLDT_ASCII_F_UP	Fixed-length single-byte character, upshifted
2	_SQLDT_DOUBLE_F	Fixed-length double-byte character
64	_SQLDT_ASCII_V	Variable-length single-byte character
65	_SQLDT_ASCII_V_UP	Variable-length single-byte character, upshifted
Numeric Data Types (128 – 134)		
130	_SQLDT_16BIT_S	16-bit signed (signed SMALLINT)

Table 3-2. Descriptor Area Data Type Declarations (page 2 of 2)

Value	Declaration in `sqlci.h` File	Description
131	_SQLDT_16BIT_U	16-bit unsigned (unsigned SMALLINT)
132	_SQLDT_32BIT_S	32-bit signed (signed INT)
133	_SQLDT_32BIT_U	32-bit unsigned (unsigned INT)
134	_SQLDT_64BIT_S	64-bit signed (signed LARGEINT)
142	_SQLDT_IEEE_REAL	32-bit floating-point (IEEE REAL)
143	_SQLDT_IEEE_DOUBLE	64-bit floating-point (IEEE DOUBLE)
Decimal Data Types (150 – 156)		
150	_SQLDT_DEC_U	Unsigned DECIMAL
151	_SQLDT_DEC_LSS	DECIMAL, leading sign separate (not SQL type)
152	_SQLDT_DEC_LSE	ASCII DECIMAL, leading sign embedded
153	_SQLDT_DEC_TSS	DECIMAL, trailing sign separate (not SQL type)
154	_SQLDT_DEC_TSE	DECIMAL, trailing sign embedded (not SQL type)
155	_SQLDT_NUM_BIG_U	Unsigned extended NUMERIC precision data type
156	_SQLDT_NUM_BIG_S	Signed extended NUMERIC precision data type
Date-Time and INTERVAL Data Types (192 – 212)		
192	_SQLDT_DATETIME	General Date-Time
195	_SQLDT_INT_Y_Y	Year to Year
196	_SQLDT_INT_MO_MO	Month to Month
197	_SQLDT_INT_Y_MO	Year to Month
198	_SQLDT_INT_D_D	Day to Day
199	_SQLDT_INT_H_H	Hour to Hour
200	_SQLDT_INT_D_H	Day to Hour
201	_SQLDT_INT_MI_MI	Minute to Minute
202	_SQLDT_INT_H_MI	Hour to Minute
203	_SQLDT_INT_D_MI	Day to Minute
204	_SQLDT_INT_S_S	Second to Second
205	_SQLDT_INT_MI_S	Minute to Second
206	_SQLDT_INT_H_S	Hour to Second
207	_SQLDT_INT_D_S	Day to Second
208	_SQLDT_INT_F_F	Fraction to Fraction
209	_SQLDT_INT_S_F	Second to Fraction
210	_SQLDT_INT_MI_F	Minute to Fraction
211	_SQLDT_INT_H_F	Hour to Fraction
212	_SQLDT_INT_D_F	Day to Fraction

Considerations for GET DESCRIPTOR

Processing Items in a Descriptor Area

You can retrieve:

- The number of filled-in descriptor items
- Fields for a specific item

You can use the number of filled-in descriptor items to construct a loop to process individual items.

Version Differences for TYPE and TYPE_FS

In NonStop SQL/MX Release 1.0, the FS type (an SQL/MX extension) was equivalent to the item TYPE, and the ANSI type was equivalent to the item TYPE_ANSI. In NonStop SQL/MX Release 1.5 and later, to comply with ANSI standards, these equivalents have changed. TYPE returns the ANSI type, and TYPE_FS returns the FS type (an SQL/MX extension).

C Examples of GET DESCRIPTOR

- Allocate a descriptor area, describe the output parameters of a dynamic SQL statement, and use the descriptor area to get information about the parameters:

```
...
desc_max = 10;
EXEC SQL ALLOCATE DESCRIPTOR 'out_sqlda' WITH MAX :desc_max;
...
EXEC SQL DESCRIBE OUTPUT dynamic_stmt
         USING SQL DESCRIPTOR 'out_sqlda';
...
/* First, get the count of the number of output values. */
EXEC SQL GET DESCRIPTOR 'out_sqlda' :num = COUNT;
/* Second, get the i-th output values and save.         */
for (i = 1; i <= num; i++) {
  EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
    :sqlda_type = TYPE,
    :sqlda_length = LENGTH,
    :sqlda_name = NAME;
  /* Test type or name to determine the host variable,  */
  /* assign data value to appropriate host variable.    */
  ...
  if (strncmp(sqlda_name,"LAST_NAME",strlen("LAST_NAME"))==0)
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
      :hv_last_name = VARIABLE_DATA;
    ...
}
...              /* process the item descriptor values */
EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqlda';
```

COBOL Examples of GET DESCRIPTOR

- Allocate a descriptor area, describe the output parameters of a dynamic SQL statement, and use the descriptor area to get information about the parameters:

```
     MOVE 10 TO desc-max.
     EXEC SQL ALLOCATE DESCRIPTOR 'out_sqlda'
                WITH MAX :desc-max END-EXEC.
     ...
     EXEC SQL DESCRIBE OUTPUT dynamic_stmt
          USING SQL DESCRIPTOR 'out_sqlda' END-EXEC.
     ...
 * First, get the count of the number of output values.
     EXEC SQL
       GET DESCRIPTOR 'out_sqlda' :num = COUNT
     END-EXEC.
 * Second, get the i-th output values and save.
     PERFORM VARYING i FROM 1 BY 1 UNTIL i > num
       EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
               :sqlda-type = TYPE,
               :sqlda-length = LENGTH,
               :sqlda-name = NAME
       END-EXEC.
 * Test type or name to determine the host variable,
 * assign data value to the appropriate host variable.
       ...
       IF sqlda-name = "LAST_NAME"
             EXEC SQL
               GET DESCRIPTOR 'out_sqlda' VALUE :i
                       :hv-last-name = VARIABLE_DATA
             END-EXEC.
       ...
     END-PERFORM.
     ...
 * Process the item descriptor values
     ...
     EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqlda' END-EXEC.
```

GET DIAGNOSTICS Statement

C/COBOL The GET DIAGNOSTICS statement returns information from the diagnostics area about the most recently executed statement and the exception status, assigns the specified statement and condition information to host variables, and stores the row number.

Use GET DIAGNOSTICS only in embedded SQL programs in C or COBOL.

```
GET DIAGNOSTICS {statement-info | condition-info}

statement-info is:

   statement-item [,statement-item]...

statement-item is:

   variable-name = statement-item-name

statement-item-name is:

     NUMBER
   | MORE
   | COMMAND_FUNCTION
   | DYNAMIC_FUNCTION
   | ROW_COUNT

condition-info is:

   EXCEPTION condition-nbr

condition-item [,condition-item]...

condition-item is:

   variable-name = condition-item-name
```

```
condition-item-name is:

     CONDITION_NUMBER
   | RETURNED_SQLSTATE
   | CLASS_ORIGIN
   | SUBCLASS_ORIGIN
   | SERVER_NAME
   | CONNECTION_NAME
   | CONSTRAINT_CATALOG
   | CONSTRAINT_SCHEMA
   | CONSTRAINT_NAME
   | CATALOG_NAME
   | SCHEMA_NAME
   | TABLE_NAME
   | COLUMN_NAME
   | CURSOR_NAME
   | MESSAGE_TEXT
   | MESSAGE_LENGTH
   | MESSAGE_OCTET_LENGTH
   | NSK_CODE
   | SQLCODE
   | ROW_NUMBER
```

statement-info

assigns statement information, *statement-info*, to host variables. See Statement Items of GET DIAGNOSTICS on page 3-57.

variable-name = statement-item-name

retrieves the named statement information item *statement-item-name* and stores the data into the named host variable *variable-name*. The data type of the target host variable must be compatible with the data type of the statement information item.

condition-info

assigns condition information, *condition-info*, to host variables. See Condition Items of GET DIAGNOSTICS on page 3-57.

EXCEPTION *condition-nbr*

specifies the number, *condition-nbr*, of the condition about which to return diagnostic information. The data type of the number is exact numeric with scale 0.

variable-name = condition-item-name

retrieves the named condition information item *condition-item-name* and stores the data into the named host variable *variable-name*. The data type of the target host variable must be compatible with the data type of the condition information item.

Statement Items of GET DIAGNOSTICS

Table 3-3 describes the statement items in the diagnostics area. For exact numeric fields with scale 0, declare the corresponding host variables as type `LONG` in C or type `PIC S9(18) COMP` in COBOL. For character fields with length greater than or equal to 128, declare the corresponding host variables as type `VARCHAR` with length 129 (with an extra byte for the null terminator) in C or type `PIC X` with length 128 in COBOL.

Table 3-3. GET DIAGNOSTICS Statement Items

Statement Item Name	Data Type and Description
NUMBER	Exact numeric, scale 0. The number of exception or completion conditions that have been stored as a result of executing the statement.
MORE	Character string, length 1. Y = more conditions were raised during execution than stored in the area. N = all the conditions raised have been stored. Reserved for future use.
COMMAND_FUNCTION	Character varying, length>=128. Identifies which SQL statement executed. Reserved for future use.
DYNAMIC_FUNCTION	Character varying, length>=128. Identifies which prepared statement executed. Reserved for future use.
ROW_COUNT	Exact numeric, scale 0. The number of rows affected by the execution of a searched DELETE or UPDATE or an INSERT. For this item, declare the corresponding host variable as type `long long` in C.

Condition Items of GET DIAGNOSTICS

Table 3-4 describes the condition items in the diagnostics area. For exact numeric fields with scale 0, declare the corresponding host variables as type `long` in C or type `PIC S9(9) COMP` in COBOL.

For character fields with length greater than or equal to 128, declare the corresponding host variables as type CHAR in C or type PIC X with length 128 in COBOL.

Table 3-4. GET DIAGNOSTICS Condition Items (page 1 of 3)

Condition Item Name	Data Type and Description
CONDITION_NUMBER	Exact numeric, scale 0. Identifies the condition.
RETURNED_SQLSTATE	Character string (5). SQLSTATE value of this condition.
CLASS_ORIGIN	Character varying, length>=128. Naming authority that defines the class value of RETURNED_SQLSTATE.
SUBCLASS_ORIGIN	Character varying, length>=128. Naming authority that defines the subclass value of RETURNED_SQLSTATE.

Table 3-4. GET DIAGNOSTICS Condition Items (page 2 of 3)

Condition Item Name	Data Type and Description
SERVER_NAME	Character varying, length>=128. Identifies server. Reserved for future use.
CONNECTION_NAME	Character varying, length>=128. Identifies connection. Reserved for future use.
CONSTRAINT_CATALOG	Character varying, length>=128. Identifies catalog of schema containing constraint or assertion. Reserved for future use.
CONSTRAINT_SCHEMA	Character varying, length>=128. Identifies schema containing constraint or assertion. Reserved for future use.
CONSTRAINT_NAME	Character varying, length>=128. Identifies constraint or assertion. Returns a fully qualified name for SQLCODE errors in the range -4000 through -4999 and in the range -8000 through -8999.
CATALOG_NAME	Character varying, length>=128. Identifies catalog of table (referenced by failed assertion) modified by statement execution. Reserved for future use.
SCHEMA_NAME	Character varying, length>=128. Identifies schema of table (referenced by failed assertion) modified by statement execution. Reserved for future use.
TABLE_NAME	Character varying, length>=128. Identifies table (referenced by failed assertion) modified by statement execution. Returns a fully qualified table name for SQLCODE errors in the range -4000 through -4999.
COLUMN_NAME	Character varying, length>=128. Identifies inaccessible column due to access rule violation. Returns a column name for SQLCODE errors in the range -4000 through -4999.
CURSOR_NAME	Character varying, length>=128. Identifies cursor in invalid state. Reserved for future use.
MESSAGE_TEXT	Character varying, length>=128. Explanatory text.
MESSAGE_LENGTH	Exact numeric, scale 0. Character length of MESSAGE_TEXT.
MESSAGE_OCTET_LENGTH	Exact numeric, scale 0. Octet length of MESSAGE_TEXT.
NSK_CODE	Exact numeric, scale 0. NSK code value of the condition.

Table 3-4. GET DIAGNOSTICS Condition Items (page 3 of 3)

Condition Item Name	Data Type and Description
SQLCODE*	Exact numeric, scale 0. SQLCODE value of this condition.
ROW_NUMBER*	Exact numeric, scale 0. Identifies the row number in error during rowset operations.

* The condition item is an SQL/MX extension.

Considerations for GET DIAGNOSTICS

Processing Condition Items in the Diagnostics Area

You can retrieve:

- The number of filled-in condition items
- Fields for a specific condition item

You can use the number of filled-in condition items to construct a loop to process individual items.

Writing a Log of Exception Conditions

You can retrieve the message text of the exception condition for SQLSTATE and SQLCODE. To provide a log of exception conditions, write the SQLSTATE and SQLCODE values, along with the message text, to a file to be used for future reference.

C Examples of GET DIAGNOSTICS

- Use the diagnostics area to get information about exception conditions:

```
...
EXEC SQL GET DIAGNOSTICS :hv_num = NUMBER,
                         ...;
for (i = 1; i <= hv_num; i++) {
  EXEC SQL GET DIAGNOSTICS EXCEPTION :i
                         :hv_sqlstate = RETURNED_SQLSTATE,
                         :hv_sqlcode = SQLCODE,
                         :hv_msgtext = MESSAGE_TEXT,
                         ...;
  /* Write to the exception condition log file. */
  ...
  /* Process the diagnostic area values. */
  ...
}
...
```

COBOL Examples of GET DIAGNOSTICS

- Use the diagnostics area to get information about exception conditions:

```
      ...
      EXEC SQL GET DIAGNOSTICS :hv-num = NUMBER,
                                ...
      END-EXEC.
      PERFORM VARYING i FROM 1 BY 1 UNTIL i > hv-num
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
                           :hv-sqlstate = RETURNED_SQLSTATE,
                           :hv-sqlcode = SQLCODE,
                           :hv-msgtext = MESSAGE_TEXT,
                           ...
        END-EXEC.
 * Write to the exception condition log file.
        ...
 * Process the diagnostic area values.
        ...
      END-PERFORM.
     ...
```

IF Statement

Considerations for IF Statement
C Example of IF Statement

C/COBOL An IF statement is a compound statement that provides conditional execution based on the truth value of a conditional expression.

IF is an SQL/MX extension that you use only in embedded SQL programs in C or COBOL.

```
IF conditional-expression THEN
   SQL-statement;[SQL-statement;]...
   [ELSEIF conditional-expression THEN
      SQL-statement;[SQL-statement;]...]...
   [ELSE SQL-statement;[SQL-statement;]...]
END IF
```

conditional-expression

specifies an SQL conditional expression. The expression can be a relational expression consisting of relational operators and more than one operand. The operands are literals or host variables combined with SQL relational operators, <, >, <=, >=, =, and <>.

The conditional expression can also contain logical operators, such as AND, OR, and NOT, and predicates but cannot contain column references or subqueries. See Search Condition on page 6-108.

The conditional expression evaluates to either true, false, or NULL.

SQL-statement;[*SQL-statement*;]...

is an SQL statement list following the THEN or ELSE keyword. The statements are executed in sequential order as in compound statement execution; the result of executing the statement list is exactly the same result as executing the statements one at a time in sequential order.

NonStop SQL/MX executes an IF statement by evaluating the first (and possibly only) *conditional-expression*. If the expression evaluates to true, NonStop SQL/MX executes the statements following the THEN keyword. If the expression evaluates to false or NULL, NonStop SQL/MX branches to the first ELSEIF part of the statement if an ELSEIF exists. Otherwise, with no ELSEIF, NonStop SQL/MX executes only the ELSE part of the statement, if there is one.

If the *conditional-expression* of the first ELSEIF evaluates to false or NULL, NonStop SQL/MX branches to the next ELSEIF part of the statement, and so on. If all the expressions evaluate to false or NULL, NonStop SQL/MX executes only the ELSE part of the statement, if there is one.

Considerations for IF Statement

SQL Statements in the List

The restrictions for which SQL statements can be used in a list are the same as the restrictions for the compound statement. Transactional SQL statements BEGIN WORK, COMMIT WORK, ROLLBACK WORK, and SET TRANSACTION cannot be used in IF statements. UPDATE STATISTICS and CONTROL statements cannot be used in IF statements.

SELECT INTO (retrieving only one row) can be used in a list. Cursors are not allowed in compound statements. However, rowsets can be used within compound statements to retrieve multiple rows from database tables.

C Example of IF Statement

- These INSERT and SELECT statements execute sequentially for new orders. Otherwise, the SELECT statement returns information about the current customer:

```
...
EXEC SQL
BEGIN
IF :hv_new_ordernum <> 0
THEN
  INSERT INTO SALES.ORDERS
    (ORDERNUM, ORDER_DATE, DELIV_DATE, SALESREP, CUSTNUM)
     VALUES (:hv_new_ordernum, :hv_orderdate, :hv_delivdate,
             :hv_salesrep, :hv_custnum);
  SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
     INTO :hv_custnum, :hv_custname,
          :hv_street, :hv_city, :hv_state, :hv_postcode
     FROM SALES.CUSTOMER
     WHERE CUSTNUM = :hv_custnum;
ELSE
  SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
    INTO :hv_custnum, :hv_custname,
         :hv_street, :hv_city, :hv_state, :hv_postcode
    FROM SALES.CUSTOMER
    WHERE CUSTNUM = :hv_current_custnum;

END IF;
END;
...
```

COBOL Example of IF Statement

- These INSERT and SELECT statements execute sequentially for new orders. Otherwise, the SELECT statement returns information about the current customer:

```
EXEC SQL
   BEGIN
   IF :hv_now_ordernum <> 0
   THEN
    INSERT INTO SALES.ORDERS
    (ORDERNUM, ORDER_NAME, DELIV_DATE,...)
    VALUES (:hv_new_ordernum, :hv_orderdate, :hv_delivdate,
            :hv_salesrep, :hv_custnum);
    SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
     INTO :hv_custnum, :hv_custname,
          :hv_street, :hv_city, :hv_state, :hv_postcode
     FROM SALES.CUSTOMER
     WHERE CUSTNUM = :hv_custnum;
   ELSE
    SELECT ...
     INTO ...
     WHERE CUSTNUM = :hv_current_custnum;
   END IF;
   END
END-EXEC.
```

INVOKE Directive

Considerations for INVOKE
C Examples of INVOKE
COBOL Examples of INVOKE

C/COBOL The INVOKE preprocessor directive generates a C structure template or COBOL record declaration that corresponds to a row in a specified table or view and inserts the declaration directly into the host program. The row description includes a data item for each column.

INVOKE is an SQL/MX extension.

```
INVOKE table
   [AS record]
   [DATEFORMAT {DEFAULT | EUROPEAN | USA}]
   [PREFIX indicator-prefix]
   [SUFFIX indicator-suffix]
   [NULL STRUCTURE]
   [CHAR AS {STRING|ARRAY}]
```

table

is the name of an existing table or view for which to create a row description. *table* can be one of these object names:

- Guardian physical name
- ANSI logical name
- DEFINE name

See Database Object Names on page 6-13.

For more information on how the preprocessor expands *table* in the INVOKE directive, see Preserving or Overriding the INVOKE Directive on page 3-66. For more information on using a DEFINE name in the Windows NT environment, see Using DEFINE Names in the Windows NT Environment on page 3-66.

AS *record*

specifies a host language identifier that is the name for the record definition or structure declaration.

For C, the default structure name is the simple name of the table or view with the suffix `_type` appended; for example: `mytable_type`.

`DATEFORMAT {DEFAULT | EUROPEAN | USA}`

specifies the format of host variables for datetime columns.

For a column with a datetime data type that has an HOUR field, DATEFORMAT USA causes INVOKE to produce a host variable that is three bytes longer than an equivalent host variable for `EUROPEAN` or `DEFAULT` format. The extra bytes allow room for "`am`" or "`pm`" following the values.

The default is DATEFORMAT DEFAULT.

`PREFIX` *`indicator-prefix`* `or SUFFIX` *`indicator-suffix`*

specifies a prefix, a suffix, or both for indicator variable names, in the form:

`indicator-prefix variable-name indicator-suffix`

The *`variable-name`* is the name of the column. If you do not specify a prefix, indicator variable names have no prefix. If you specify a prefix but do not specify a suffix, indicator names have no suffix. If you do not specify either a prefix or a suffix, the suffix depends on the language, as follows:

```
C       _i
COBOL   -I
```

A prefix or suffix must consist of legal identifier values for the host language in which it is used. However, you can use uppercase or lowercase letters in a prefix or suffix, regardless of the host language. For C, INVOKE makes the suffix lowercase.

If the indicator variable name with suffix is longer than 31 characters, the name is truncated to 31 characters. A warning is issued for each truncated name.

`NULL STRUCTURE`

specifies that a column allowing null should be declared as a structure with the same name as the column and with fields for the data item and its indicator variable. The fields are named `INDICATOR` and `VALU`.

`CHAR AS {STRING | ARRAY}`

(for C programs) specifies whether to create a byte for the null terminator in C character types:

`STRING`	Generate the extra byte.
`ARRAY`	Omit the extra byte.

The default is CHAR AS STRING.

For more information about creating C and COBOL host variables using INVOKE, see the *SQL/MX Programming Manual for C and COBOL*.

Considerations for INVOKE

Preserving or Overriding the INVOKE Directive

The preprocessor expands the table or view name in the INVOKE directive to a fully qualified object name, such as `[\`*`node`*`.][[$`*`volume`*`.]`*`subvol.`*`]`*`filename`* for a Guardian physical name or *`catalog`*`.`*`schema`*`.`*`object`* for an ANSI logical name. Expansion of the name depends on how the name is qualified in the INVOKE directive and how you have declared object names in the program. The preprocessor either overrides the name in the INVOKE directive with the object name declaration or preserves the qualified name in the INVOKE directive.

For more information on whether the preprocessor preserves or overrides the INVOKE directive for Guardian physical names, see DECLARE MPLOC Declaration on page 3-29. For more information on whether the preprocessor preserves or overrides the INVOKE directive for ANSI logical names, see DECLARE NAMETYPE Declaration on page 3-32 and DECLARE SCHEMA Declaration on page 3-33.

Using DEFINE Names in the Windows NT Environment

To preprocess an embedded SQL program that uses a DEFINE in the INVOKE directive in the Windows NT environment, you must set an environment variable for the DEFINE:

1. At the Windows NT prompt, type:

   ```
   set tab_envar =[\node.]$vol.subvol.table
   ```

2. In the INVOKE directive, use the name of the environment variable as the DEFINE name:

   ```
   EXEC SQL INVOKE =tab_envar AS tab_type;
   ```

The preprocessor then expands the INVOKE directive:

```
EXEC SQL INVOKE [\node.]$vol.subvol.tablename AS tab_type;
```

SYSKEY Column

If the table that you specify in the INVOKE directive does not contain a user-defined primary key, the INVOKE directive includes the SYSKEY column in the structure or record that it generates. Otherwise, SYSKEY is omitted from the INVOKE-generated structure or record.

Authorization Requirements

To use INVOKE on a table or view, you must have SELECT privileges on all the columns.

Using INVOKE in a C Program

The general syntax for using an embedded INVOKE directive within an SQL declare section in a C program is:

```
EXEC SQL INVOKE table [AS structure-name];
struct structure-name structure-instance ;
```

The `struct` declaration declares *`structure-instance`* to be a structure of the type *`structure-name`*. You must declare a variable of the `struct` type so that you can use that variable in your C language statements.

Using `typedef` for a Structure

You can use `typedef` to create your own name for a structure that is created with an INVOKE directive. A `typedef struct` statement can be global or local in scope and must be defined in the SQL declare section of a C program.

C Examples of INVOKE

- Suppose that the EMPLOYEE table consists of the EMPNUM, FIRST_NAME, LAST_NAME, and DEPTNUM columns. The FIRST_NAME column allows null, and the EMPNUM column is the primary key. This example shows an INVOKE statement with the NULL STRUCTURE clause and the generated structure template:

```
EXEC SQL BEGIN DECLARE SECTION;
  EXEC SQL DECLARE SCHEMA 'samdbcat.persnl';
  EXEC SQL INVOKE employee AS emptbl_rec NULL STRUCTURE;
  struct emptbl_rec,  emptbl_rec1, emptbl_rec2;
  ...
EXEC SQL END DECLARE SECTION;
/* Input employee number for search */
...
EXEC SQL
  SELECT empnum, first_name, last_name, deptnum
  INTO :emptbl_rec1.empnum, :emptbl_rec1.first_name,
       :emptbl_rec1.last_name, :emptbl_rec1.deptnum
  FROM persnl.employee
  WHERE empnum = :hv_this_employee;
```

The SQL/MX C preprocessor generates the structure immediately after the INVOKE statement in the preprocessed program code:

```
/*  Beginning of generated code for SQL INVOKE */
 struct emptbl_rec {
   unsigned short   empnum;
   struct {
     short          indicator;
     CHAR           valu[16];
   } first_name;
   CHAR             last_name[21];
   unsigned short   deptnum;
 };
```

- This example shows the generated structure template that the previous INVOKE directive would have generated if the EMPLOYEE table did not contain a user-defined primary key, such as the EMPNUM column. Note the presence of SYSKEY:

```
/*  Beginning of generated code for SQL INVOKE */
 struct emptbl_rec {
   long long        syskey;
   unsigned short   empnum;
   struct {
     short          indicator;
     CHAR           valu[16];
   } first_name;
   CHAR             last_name[21];
   unsigned short   deptnum;
 };
```

- Use a Guardian physical name for the table in the INVOKE directive. You must explicitly declare the NAMETYPE as NSK for Guardian physical names, because the default NAMETYPE is ANSI:

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL DECLARE NAMETYPE 'NSK';
EXEC SQL DECLARE MPLOC '$data07.persnl';
EXEC SQL INVOKE employee AS emptbl_rec NULL STRUCTURE;
struct emptbl_rec emptbl_rec1, emptbl_rec2;
...

EXEC SQL END DECLARE SECTION;
```

- Invoke an SQL table named `=classdef` and refer to the structure by the identifier `classdef_type`. Use a `typedef struct` statement to define `CLASSDEF` as the name of the structure type for the variable `row`. Use the `row` variable to access rows of data from the table:

```
#pragma section classdef

EXEC SQL BEGIN DECLARE SECTION;
  EXEC SQL INVOKE =classdef AS classdef_type;
typedef struct classdef_type  CLASSDEF;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL BEGIN DECLARE SECTION;
  CLASSDEF row;
EXEC SQL END DECLARE SECTION;
```

COBOL Examples of INVOKE

- Suppose that the EMPLOYEE table consists of the EMPNUM, FIRST_NAME, LAST_NAME, and DEPTNUM columns. The FIRST_NAME column allows null, and the EMPNUM column is the primary key. This example shows an INVOKE statement with the NULL STRUCTURE clause and part of the structure that is generated:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  EXEC SQL INVOKE employee AS EMPTBL-REC NULL STRUCTURE
  END-EXEC.
EXEC SQL END DECLARE SECTION END-EXEC.
```

  NonStop SQL/MX generates this record:

```
*  Record Definition for table PERSNL.EMPLOYEE
   01 EMPTBL-REC.
      02 EMPNUM        PIC 9(4) comp.
      02 FIRST-NAME.
         03 INDICATOR  PIC S9(4) comp.
         03 VALU       PIC X(15).
      02 LAST-NAME     PIC X(20).
      02 DEPTNUM       PIC 9(4) comp.
```

- This example shows the generated structure template that the previous INVOKE directive would have generated if the EMPLOYEE table did not contain a user-defined primary key, such as the EMPNUM column. Note the presence of SYSKEY:

```
*  Record Definition for table PERSNL.EMPLOYEE
   01 EMPTBL-REC.
      02 SYSKEY        PIC S9(18) comp.
      02 EMPNUM        PIC 9(4) comp.
      02 FIRST-NAME.
         03 INDICATOR  PIC S9(4) comp.
         03 VALU       PIC X(15).
      02 LAST-NAME     PIC X(20).
      02 DEPTNUM       PIC 9(4) comp.
```

MODULE Directive

The MODULE directive specifies the name of an embedded SQL/MX module for the C or COBOL preprocessor. If you do not specify a MODULE directive, the preprocessor generates a module name.

MODULE is an SQL/MX extension that you use only in embedded SQL programs.

```
MODULE module-name [NAMES ARE ISO88591]
```

module-name

is the name of the module. NonStop SQL/MX automatically qualifies a module name with the current default catalog and schema names unless you explicitly specify catalog and schema names with the module name. The module name is an SQL identifier and must be unique among module names in the schema.

For a module name of `catalog.schema.name`, *catalog*, *schema*, and *name* are SQL identifiers and therefore cannot consist of more than 128 characters. See Identifiers on page 6-56.

△ **Caution.** Avoid using delimited identifiers that contain dots (.) and trailing spaces in the names of the catalog, schema, and module. Dots and trailing spaces in delimited identifiers might cause the three-part module name to clash with an unrelated module name, thus overwriting the query execution plans of the other module.

NAMES ARE ISO88591

is an optional clause that specifies the character set for the module as ISO88591.

The ISO88591 character set is the default character set for CHAR or VARCHAR data types for NonStop SQL/MX. The ISO 8859 character sets are a standard set of nine single-byte character sets defined by ISO (International Organization for Standardization) in a series called ISO 8859. The ISO88591 character set supports English and other Western European languages.

Considerations for MODULE

Directive Used by the Preprocessor

C/COBOL

The 3GL preprocessor creates a module definition file, containing only SQL statements, as one of its two output files. The preprocessor writes the header of the module definition file as:

```
MODULE module-name NAMES ARE ISO88591 ;
TIMESTAMP DEFINITION ( creation_timestamp ) ;
```

The preprocessor gets the *module-name* from the MODULE directive, if one exists, at the beginning of your embedded SQL 3GL program. ■

Automatic Generation of Module Names

If you do not specify a MODULE directive, the preprocessor or customizer generates a module name for you. If you change your source program and process and compile it again, the new module overwrites the old module. System-generated module names can become a management problem if you want to create different versions of your program. For more information on module management, see the *SQL/MX Programming Manual for C and COBOL*.

C Examples of MODULE

- This example shows a MODULE directive:

  ```
  EXEC SQL MODULE EXF61M NAMES ARE ISO88591;
  ```

COBOL Examples of MODULE

- This example shows a MODULE directive:

  ```
  EXEC SQL MODULE EXF62M NAMES ARE ISO88591 END-EXEC.
  ```

OPEN Statement

Considerations for OPEN
C Examples of OPEN
COBOL Examples of OPEN

C/COBOL

The OPEN statement opens a cursor in a host program and establishes the result table specified by the DECLARE CURSOR statement for the named cursor. It positions the cursor before the first row of the result table.

In dynamic SQL, the cursor name is provided at execution time. An optional USING clause provides input parameters for the cursor specification. Otherwise, there is no difference between the static and dynamic forms of OPEN.

Use OPEN only in embedded SQL programs in C or COBOL.

```
OPEN {cursor-name  | ext-cursor-name}
   [USING {argument-list | descriptor-spec}]

ext-cursor-name is:
   [GLOBAL | LOCAL] value-specification

argument-list is:
   variable-spec [,variable-spec]...

descriptor-spec is:
   SQL DESCRIPTOR descriptor-name

variable-spec is:
   :variable-name [[INDICATOR] :indicator-name]

descriptor-name is:
   [GLOBAL | LOCAL] value-specification
```

`cursor-name`

is an SQL identifier—the name of a cursor. The cursor must be previously declared and not already open. See Identifiers on page 6-56.

`ext-cursor-name`

is a `value-specification`—a character literal or a host variable with character data type. When OPEN executes, the content of the host variable (if used) gives the name of the cursor. The cursor must be previously declared and not already open.

`GLOBAL | LOCAL`

specifies scope. The default is LOCAL. The scope of a GLOBAL cursor is the SQL session. The scope of a LOCAL cursor is the module or compilation unit in which OPEN appears.

`USING` *`variable-spec`* `[,`*`variable-spec`*`]...`

(dynamic SQL) identifies the host variables for the dynamic input parameters of the cursor specification.

Before OPEN with USING executes, the application must store information for each parameter of the cursor specification in the appropriate host variable.

`USING SQL DESCRIPTOR` *`descriptor-name`*

(dynamic SQL) identifies the SQL descriptor area for the dynamic input parameters of the cursor specification. An SQL descriptor area must be currently allocated whose name is the value of *`descriptor-name`* and whose scope is the same scope as specified in the OPEN statement.

Before OPEN with USING executes, the application must store information for each input parameter of the cursor specification in the descriptor area. Each parameter has an item descriptor.

Considerations for OPEN

Establishing the Result Table

If the cursor specification includes embedded variables, the variables are evaluated when OPEN executes. Any subsequent changes to those variables do not affect the result table of the cursor specification.

Authorization Requirements

To execute OPEN, you must have read authority for tables or views referred to in the SELECT associated with the cursor. If the cursor was declared FOR UPDATE, you must also have write authority to the tables.

Declaring Host Variables

The host variables occurring in the cursor specification must be declared within the scope of the OPEN statement. Otherwise, an error occurs during preprocessing.

Using Extended Dynamic Cursors

The name of an extended dynamic cursor is not known until run time. When OPEN executes, the name must identify an allocated cursor within the same scope.

USING Clause

If the cursor specification uses dynamic input parameters, you must provide a USING clause for either a list of arguments or an SQL descriptor area. This requirement is the same as that for providing a USING clause for an EXECUTE statement that executes a prepared statement with dynamic input parameters. See EXECUTE Statement on page 2-201.

C Examples of OPEN

- Declare and open a cursor, using FETCH to retrieve data, then closing the cursor:

```
EXEC SQL DECLARE cursor1 CURSOR FOR
         SELECT  COL1, COL2, COL3
         FROM    PARTS
         WHERE   COL1  >= :hostvar
         ORDER BY COL1
         READ UNCOMMITTED ACCESS;
...
EXEC SQL OPEN cursor1;
...
EXEC SQL FETCH cursor1
         INTO :hostvar1, :hostvar2, :hostvar3;
...
EXEC SQL CLOSE cursor1;
```

- This example uses extended cursor and statement names in the PREPARE, ALLOCATE CURSOR, and OPEN statements:

```
...
scanf("%s", in_curspec);
...
EXEC SQL PREPARE :curspec FROM :in_curspec;
...
EXEC SQL ALLOCATE :extcur CURSOR FOR :curspec;
...
EXEC SQL OPEN :extcur;
...
```

COBOL Examples of OPEN

- Declare and open a cursor, using FETCH to retrieve data, then closing the cursor:

```
EXEC SQL DECLARE cursor1 CURSOR FOR
         SELECT  COL1, COL2, COL3
         FROM    PARTS
         WHERE   COL1  >= :hostvar
         ORDER BY COL1
         READ UNCOMMITTED ACCESS
END-EXEC.
...
EXEC SQL OPEN cursor1 END-EXEC.
...
EXEC SQL FETCH cursor1
         INTO :hostvar1, :hostvar2, :hostvar3
END-EXEC.
...
EXEC SQL CLOSE cursor1 END-EXEC.
```

- This example uses extended cursor and statement names in the PREPARE, ALLOCATE CURSOR, and OPEN statements:

```
...
ACCEPT in-curspec.
...
EXEC SQL PREPARE :curspec FROM :in-curspec END-EXEC.
...
EXEC SQL ALLOCATE :extcur CURSOR FOR :curspec END-EXEC.
...
EXEC SQL OPEN :extcur END-EXEC.
...
```

SET (Assignment) Statement

C Examples of Assignment Statement

C/COBOL An assignment statement in a compound statement assigns a value to a host variable so that subsequent statements in the containing compound statement can reference and use the value of that host variable.

SET is an SQL/MX extension that you use only in embedded SQL programs in C or COBOL.

```
SET assignment-target = assignment-source

assignment-target is:
   variable-spec [,variable-spec]...

assignment-source is:
     subquery
   | {expression | rowset-expression | NULL}
   [,{expression | rowset-expression | NULL]...
```

assignment-target

specifies a list of host variable specifications in which to return the values in the *assignment-source*.

The number of items in *assignment-source* must be equal to the number of specified host variables, and the data type of each source value must be compatible with the data type of its target host variable. The first value in the *assignment-source* is assigned to the first host variable, the second value to the second variable, and so on.

You can use rowset host variables as *assignment-targets* if either the *assignment-source* is a subquery that returns more than one row, or the *assignment-source* includes *rowset-expressions*. In this case you can use rowset host variables in the *assignment-target* to return values from multiple consecutive rows in the *assignment-source*.

The values of the host variables that have been set by an assignment statement are made available for use by subsequent statements within the compound statement and by other statements that follow after that compound statement.

:*variable-name* [[INDICATOR] :*indicator-name*]

is a variable specification—a host variable with optionally an indicator variable. A variable name begins with a colon (:).

assignment-source

specifies a list of value expressions, a list of rowset expressions, the NULL specification, or a subquery. These values are to be inserted into the *assignment-target.*

```
  subquery
| {expression | rowset-expression | NULL}
[,{expression | rowset-expression | NULL}]...
```

is a value expression, rowset expressions, NULL, or the result of a subquery.

NULL can be assigned to a host variable of any type.

If you use a rowset expression or if the subquery returns more than one row, the assignment target must consist of rowset host variables.

C Examples of Assignment Statement

- This SET statement inside the BEGIN and END keywords sets a value that is used in the INSERT statement:

```
...
EXEC SQL
BEGIN
SET :hv_salesrep = 999;  /* sales rep is not known /*
INSERT INTO SALES.ORDERS
     (ORDERNUM, ORDER_DATE, DELIV_DATE, SALESREP, CUSTNUM)
      VALUES (:hv_ordernum, :hv_orderdate, :hv_delivdate,
              :hv_salesrep, :hv_custnum);
END;
...
```

SET DESCRIPTOR Statement

SQL Item Descriptor Area of SET DESCRIPTOR
Note.
C Examples of SET DESCRIPTOR
COBOL Examples of SET DESCRIPTOR

C/COBOL

The SET DESCRIPTOR statement changes specified information in an SQL descriptor area. An application program can either set the count of item descriptors with information or set the value of a specific field within a specific item.

Use SET DESCRIPTOR only in embedded SQL programs in C or COBOL.

```
SET DESCRIPTOR descriptor-name {set-descriptor-info}

descriptor-name is:
   [GLOBAL | LOCAL] value-specification

set-descriptor-info is:
     COUNT = value-specification
   | ROWSET_SIZE = value-specification
   | VALUE item-number set-item-info [,set-item-info]...

set-item-info is:
   descriptor-item-name = value-specification

descriptor-item-name is:
     CHARACTER_SET_NAME
   | DATETIME_CODE
   | INDICATOR_DATA | INDICATOR
   | INDICATOR_POINTER
   | INDICATOR_TYPE
   | LEADING_PRECISION
   | LENGTH
   | PRECISION
   | ROWSET_IND_LAYOUT_SIZE
   | ROWSET_VAR_LAYOUT_SIZE
   | SCALE
   | TYPE
   | TYPE_FS
   | VARIABLE_DATA | DATA
   | VARIABLE_POINTER
```

descriptor-name

is a *value-specification*—a character literal or host variable with character data type. The named SQL descriptor area must be currently allocated.

COUNT = *value-specification*

sets the COUNT of item descriptors from a *value-specification*—a literal with exact numeric data type or a host variable. *value-specification* is the number of dynamic parameters described in the descriptor area.

`ROWSET_SIZE =` *`value-specification`*

sets the length of rowsets specified in the item descriptors from *`value-specification`*, a literal with exact numeric data type or a host variable. *`value-specification`* is the common length of all rowsets described in the descriptor.

`VALUE item-number set-item-info [,set-item-info]...`

sets the value of a specific field within a specific item. See SQL Item Descriptor Area of SET DESCRIPTOR on page 3-79.

`item-number`

refers to a particular item in the SQL descriptor area. *`item-number`* must be a host variable. The data type of *`item-number`* must be exact numeric, and its value must be less than or equal to the maximum number of occurrences specified when the SQL descriptor area was allocated. If *`item-number`* exceeds the value of COUNT, a completion condition is raised (no data). See ALLOCATE DESCRIPTOR Statement on page 3-6.

`descriptor-item-name = value-specification`

specifies the field *`descriptor-item-name`* in which to store the information and the *`value-specification`* that is or contains the information—a literal or a host variable.

SQL Item Descriptor Area of SET DESCRIPTOR

Table 3-5 describes the items in the descriptor area for SET DESCRIPTOR. For exact numeric fields with scale 0, declare the corresponding host variables as type `long` in C and `PIC S9(9) COMP` in COBOL. For character fields, declare the corresponding host variables as type `VARCHAR` in C (with an extra byte for the null terminator) and type `PIX X` in COBOL.

Table 3-5. SET DESCRIPTOR Descriptor Area Items (page 1 of 2)

Field	Data Type and Description
CHARACTER_SET_NAME	Character string. One-part character set name.
DATETIME_CODE	Exact numeric, scale 0. Codes for DATETIME type: 1 date; 2 time; 3 timestamp. Codes for INTERVAL subfields: 1 year; 2 month; 3 day; 4 hour; 5 minute; 6 second; 7 year to month; 8 day to hour; 9 day to minute; 10 day to second; 11 hour to minute; 12 hour to second; 13 minute to second. This field is equivalent to the ANSI-named DATETIME_INTERVAL_CODE field. You cannot use the ANSI name for this field.**
INDICATOR_DATA	Exact numeric, scale 0. Value for the indicator variable of VARIABLE_DATA: 0 INDICATOR_DATA is not null. <0 INDICATOR_DATA is null. >0 VARIABLE_DATA is truncated and INDICATOR_DATA is the length of the source data. This field is equivalent to the ANSI-named INDICATOR field. You can also use INDICATOR as the name of the field.**
INDICATOR_TYPE*	Exact numeric, scale 0. Type of INDICATOR_DATA. The default type is `short`. Values for INDICATOR_TYPE are: -1 numeric data is negative 0 (optional) numeric data is positive
LEADING_PRECISION	Exact numeric, scale 0. Precision of interval start field. This field is equivalent to the ANSI-named DATETIME_INTERVAL_PRECISION field. You cannot use the ANSI name for this field.**
LENGTH	Exact numeric, scale 0. Length (in characters) for strings.
OCTET_LENGTH	Exact numeric, scale 0. Length in bytes for strings.
PRECISION	Exact numeric, scale 0. Precision for numeric types. PRECISION specifies the total number of digits and cannot exceed 18.
ROWSET_IND_LAYOUT_SIZE	Exact numeric, scale 0. Size of an individual array element in a rowset host variable. A value 0 (zero) in this field denotes that the host variable is not of rowset type and is a scalar host variable.
ROWSET_VAR_LAYOUT_SIZE	Exact numeric, scale 0. Size of an individual array element in a rowset host variable. A value 0 (zero) in this field denotes that the host variable is not of rowset type and is a scalar host variable.

* The statement item is an SQL/MX extension.

** The SQL/MX name is different from the ANSI name.

Table 3-5. SET DESCRIPTOR Descriptor Area Items (page 2 of 2)

Field	Data Type and Description
SCALE	Exact numeric, scale 0. Scale for exact numeric types. SCALE specifies the number of digits to the right of the decimal point.
TYPE	Exact numeric, scale 0. ANSI codes for data type: 1 CHARACTER; 2 NUMERIC; 3 DECIMAL; 4 INTEGER; 5 SMALLINT; 6 IEEE FLOAT; 7 IEEE REAL; 8 DOUBLE PRECISION; 9 DATE, TIME, or TIMESTAMP; 10 INTERVAL; 12 CHARACTER VARYING. SQL/MX extensions: -101 character uppercase; -201 numeric unsigned; -301 decimal unsigned; -302 decimal large; -303 decimal large unsigned; -401 integer unsigned; -402 largeint; -502 smallint unsigned; -601 character varying with length specified in first two bytes. See Version Differences for TYPE and TYPE_FS on page 3-84.
TYPE_FS*	Exact numeric, scale 0. These codes are the SQL/MP-specific codes returned in the SQL descriptor area, as shown under SQL Descriptor Area Data Type Declarations of GET DESCRIPTOR on page 3-51. This field does not provide the ANSI codes for data type. The ANSI codes are provided by TYPE. See Version Differences for TYPE and TYPE_FS on page 3-84.
VARIABLE_DATA	Actual data associated with the dynamic parameter. The type, length, name, and so on, are determined by other fields. This field is equivalent to the ANSI-named DATA field. You can also use DATA as the name of the field.** This field cannot contain arithmetic computation.
VARIABLE_POINTER*	Pointer to the value of VARIABLE_DATA.

* The statement item is an SQL/MX extension.
** The SQL/MX name is different from the ANSI name.

Note. You must declare the host variables for the exact numeric fields in the above table as 32-bit integers. However, declare the host variables as long for VARIABLE_POINTER and INDICATOR_POINTER.

Considerations for SET DESCRIPTOR

When you set the TYPE item, all other items are reset to the default. If you set TYPE, you must specify information for any other items you need.

Null values and SET DESCRIPTOR

If you use SET DESCRIPTOR to set a column's value to be a null value, you must initialize the host variable to a number with INDICATE_DATA = -1. Otherwise, NonStop SQL/MX issues an error. For example:

```
unsigned DECIMAL(3,3) data_pic_2;
strncpy(data_pic_2, " ", sizeof(data_pic_2));
strcpy(data_pic_2, " 000");
col_num = 1;
data_ind = -1;

EXEC SQL SET DESCRIPTOR :upd_desc VALUE :col_num
     INDICATOR_DATA = :data_ind,
     VARIABLE_DATA = :data_pic_2;
printf("SQLCODE after SET DESCRIPTOR is %d \n", SQLCODE);
EXEC SQL EXECUTE P_UPD USING SQL DESCRIPTOR :upd_desc;
printf("SQLCODE after EXECUTE = %ld\n", SQLCODE);
```

DECIMAL Data Types and SET DESCRIPTOR

When a host variable is defined as the DECIMAL data type in an embedded program, there are two ways to set the descriptor:

- Set with the DECRIBE statement:

```
    strncpy(insert_buf, " ", sizeof(insert_buf));
    strcpy(insert_buf, "INSERT INTO t ( decimal_3_unsigned)
values (cast (? as decimal(3,0) unsigned));");

    strncpy(decimal_3_unsigned, " ",
sizeof(decimal_3_unsigned));
    strcpy(decimal_3_unsigned, " 382");

    desc_max = 1;

    EXEC SQL ALLOCATE DESCRIPTOR 'in_desc2' WITH MAX
:desc_max;
    printf("SQLCODE after allocate descriptor  is %d\n",
 SQLCODE);

    EXEC SQL PREPARE FROM :insert_buf;
    printf("SQLCODE after prepare is %d\n", SQLCODE);

    EXEC SQL DESCRIBE INPUT USING SQL DESCRIPTOR 'in_desc2';
    printf("SQLCODE after descriptor is %d\n", SQLCODE);

    desc_val = 1;
    type_val = -301;

    EXEC SQL SET DESCRIPTOR 'in_desc2' VALUE :desc_val
                 DATA = :decimal_3_unsigned;
    printf("SQLCODE after SET DESCRIPTOR is %d\n",
SQLCODE);
```

```
    EXEC SQL EXECUTE USING SQL DESCRIPTOR
'in_desc2';
    printf("SQLCODE after insert is %d\n", SQLCODE);
```

- Set without the DECRIBE statement. You must set LENGTH or the descriptor will receive a numeric overflow error.

```
    strncpy(insert_buf, " ", sizeof(insert_buf));
    strcpy(insert_buf, "INSERT INTO t1 (decimal_3_unsigned) "
             " VALUES (cast(? as decimal(3,0) unsigned));");

    desc_max = 1;

    EXEC SQL ALLOCATE DESCRIPTOR 'in_desc3' WITH MAX
:desc_max;
    printf("SQLCODE after allocate descriptor is %d\n",
SQLCODE);

    EXEC SQL PREPARE FROM :insert_buf;
    printf("SQLCODE after prepare is %d\n", SQLCODE);

    desc_val = 1;
    type_val = -301;
    strncpy(decimal_3_unsigned, " ",
sizeof(decimal_3_unsigned));
    strcpy(decimal_3_unsigned, " 999");
    precision = 3;
    scale = 0;
    length = 5;

    EXEC SQL SET DESCRIPTOR 'in_desc3' VALUE :desc_val
                 TYPE = :type_val,
                 PRECISION = :precision,
                 SCALE = :scale,
                 LENGTH = :length,
                 DATA = :decimal_3_unsigned;

    printf("SQLCODE after SET DESCRIPTOR is %d\n", SQLCODE);
    EXEC SQL EXECUTE USING SQL DESCRIPTOR 'in_desc3';
    printf("SQLCODE after insert r1 is %d\n", SQLCODE);
```

Using VARIABLE_POINTER

If the VARIABLE_POINTER value is set in the descriptor area, the type and length of the host variable pointed to must exactly match the type and length of the corresponding item in the descriptor area. The type and length of the item in the descriptor area is set either by executing a DESCRIBE INPUT statement or by setting the TYPE and LENGTH items in the descriptor area.

If the type and length are not identical, the results can be unpredictable at program execution time. To avoid this problem, use one of these alternatives:

- Use arguments in an EXECUTE statement rather than descriptor areas. For more information about dynamic SQL, see the *SQL/MX Programming Manual for C and COBOL*.
- If descriptor areas are used, use VARIABLE_DATA rather than VARIABLE_POINTER.

Use VARIABLE_POINTER to efficiently retrieve individual values from a large buffer. For more information on retrieving multiple values from a large buffer, see the *SQL/MX Programming Manual for C and COBOL*.

VARIABLE_POINTER is not supported in COBOL. Embedded COBOL does not support the pointer type.

Processing Items in a Descriptor Area

You can retrieve the number of filled-in descriptor items for input parameters and set fields for a specific item. Use the number of filled-in descriptor items to construct a loop to set values for individual parameters.

When a DESCRIBE statement executes, NonStop SQL/MX identifies parameters by the context in which they appear in a prepared statement.

If you execute a DESCRIBE statement before a SET DESCRIPTOR statement, you need not include TYPE in the SET DESCRIPTOR statement.

Version Differences for TYPE and TYPE_FS

In SQL/MX Release 1.0, the FS type (an SQL/MX extension) was equivalent to the item TYPE, and the ANSI type was equivalent to the item TYPE_ANSI. In SQL/MX Release 1.5 and later, to comply with ANSI standards, these equivalents have changed. TYPE returns the ANSI type, and TYPE_FS returns the FS type (an SQL/MX extension).

C Examples of SET DESCRIPTOR

- Allocate a descriptor area, describe the input parameters of a dynamic SQL statement, and use the descriptor area to set values for the parameters:

```
desc_max = 1;
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc_max;
...
strcpy (hv_sql_statement, "UPDATE employee"
        " SET salary = salary * 1.1"
        " WHERE jobcode = CAST(? AS NUMERIC(4) unsigned)");
...
EXEC SQL PREPARE sqlstmt FROM :hv_sql_statement;
EXEC SQL DESCRIBE INPUT sqlstmt
         USING SQL DESCRIPTOR 'in_sqlda';
...
scanf("%ld",&in_jobcode);
desc_value = 1;
```

```
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
  VARIABLE_DATA = :in_jobcode,
  ...;
EXEC SQL EXECUTE sqlstmt USING SQL DESCRIPTOR 'in_sqlda';
```

COBOL Examples of SET DESCRIPTOR

- Allocate a descriptor area, describe the input parameters of a dynamic SQL statement, and use the descriptor area to set values for the parameters:

```
     MOVE 1 TO desc-max.
     EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda'
            WITH MAX :desc-max END-EXEC.
     ...
     MOVE "UPDATE employee
-         " SET salary = salary * 1.1
-         " WHERE jobcode = CAST(? AS NUMERIC(4) unsigned)"
          TO hv-sql-statement.
     EXEC SQL PREPARE sqlstmt
              FROM :hv-sql-statement END-EXEC.
     EXEC SQL DESCRIBE INPUT sqlstmt
              USING SQL DESCRIPTOR 'in_sqlda' END-EXEC.
     ACCEPT in-jobcode.
     MOVE 1 TO desc-value.
     EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc-value
              VARIABLE_DATA = :in-jobcode,
              ...
     END-EXEC.
     EXEC SQL EXECUTE sqlstmt
              USING SQL DESCRIPTOR 'in_sqlda' END-EXEC.
```

WHENEVER Declaration

[Considerations for WHENEVER](#)
[C Examples of WHENEVER](#)
[COBOL Examples of WHENEVER](#)

C/COBOL The WHENEVER declarative statement specifies an action to take when an error, warning, or no-rows-found condition occurs. The preprocessor inserts code after every embedded SQL statement to check values of SQLSTATE and jump to the appropriate routine to handle the error, warning, or no-rows-found condition.

Use WHENEVER only in embedded SQL programs in C or COBOL.

```
WHENEVER condition condition-action

condition is:
   NOT FOUND | SQLERROR | SQL_WARNING

condition-action is:
     CONTINUE
   | GOTO target
   | CALL C-function
   | PERFORM COBOL-routine
```

condition

specifies the condition to test for:

NOT FOUND	A no-rows-found condition
SQLERROR	An error
SQL_WARNING	A warning

NonStop SQL/MX tests for *condition* after each DDL and DML statement for which the WHENEVER declaration is in effect. To end testing, specify WHENEVER with the same condition but no action.

In a SELECT through a cursor, NOT FOUND means no rows or all rows qualify. In statements with a WHERE clause, NOT FOUND means no rows satisfy the WHERE clause. In a FETCH after a series of fetches, NOT FOUND means all rows were fetched.

condition-action

specifies the action to take:

CONTINUE	Continue with next statement.
GOTO *target*	Pass control to the target location.
CALL *C-function*	Execute the named C function.
PERFORM *COBOL-routine*	Execute the named COBOL routine.

If you do not specify an action, NonStop SQL/MX discontinues checking for the specified condition.

target

is a host label identifier that specifies a target location in a C program.

Considerations for WHENEVER

SQL/MX Extensions to WHENEVER

The SQL_WARNING condition and the CALL action are SQL/MX extensions.

Status Codes

FETCH returns a five-character status code to SQLSTATE, whose values include:

00000	The FETCH was successful.
02000	NOT FOUND—The result table is empty or the end of the table was encountered.
22xxx	SQLERROR—Data exception condition.

FETCH also returns an integer status code to SQLCODE, whose values include:

0	The FETCH was successful.
100	NOT FOUND—The result table is empty or the end of the table was encountered.
> 0	SQL_WARNING—A warning was issued.
< 0	SQLERROR—An error occurred.

C Examples of WHENEVER

- The effect of this statement is the same as if you had written a C statement that tests for an SQLSTATE of `02000` and executes a C `continue` statement:

  ```
  WHENEVER NOT FOUND CONTINUE;
  ```

- WHENEVER sets actions for all embedded SQL statements that physically follow it in the program. In this example, if `statement_2` caused an error, control continues at label `x`.

  ```
  ...
  EXEC SQL statement_1;
  EXEC SQL WHENEVER SQLERROR GOTO label_x;
  EXEC SQL statement_2;
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL statement_3;
  ...
  label_x:
  ...
  ```

COBOL Examples of WHENEVER

For more COBOL examples, see the *SQL/MX Programming Manual for C and COBOL*.

- The effect of this statement is the same as if you had written a COBOL statement that tests for an SQLSTATE of `02000` and executes a COBOL `NEXT SENTENCE` statement:

  ```
  WHENEVER NOT FOUND CONTINUE END-EXEC.
  ```

- WHENEVER sets actions for all embedded SQL statements that physically follow it in the program. In this example, if `statement_2` caused an error, control continues at paragraph `x`:

  ```
      ...
      EXEC SQL statement_1 END-EXEC.
      EXEC SQL WHENEVER SQLERROR GOTO para-x END-EXEC.
      EXEC SQL statement_2 END-EXEC.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      EXEC SQL statement_3 END-EXEC.
      ...
  ```

4 MXCI Commands

This section describes the syntax and semantics of MXCI commands. MXCI commands are NonStop SQL/MX extensions that typically affect attributes of an MXCI session. You can run these commands only through MXCI, with the exceptions noted:

GET VERSION OF SCHEMA Command on page 4-38	Displays the schema version of the specified schema.
GET VERSION OF SYSTEM SCHEMA Command on page 4-39	Displays the system schema version of the specified node.
GET VERSION OF Object Command on page 4-40	Displays the object schema version (OSV) and the object feature version (OFV) of the specified database object.
GET VERSION OF MODULE Command on page 4-41	Displays the module version of the specified module.
GET VERSION OF PROCEDURE Command on page 4-42	Displays the plan version of the specified procedure in the specified module.
GET VERSION OF STATEMENT Command on page 4-43	Displays the plan version of the specified prepared statement.
HISTORY Command on page 4-44	Displays recently executed commands.
INFO DEFINE Command on page 4-45	Displays current DEFINEs.
INVOKE Command on page 4-46	Displays a record description for the specified table or view. Can also be run as embedded SQL to create a C structure or COBOL record. See INVOKE Directive on page 3-64.
LOG Command on page 4-47	Starts or ends session logging to a file.
LS Command on page 4-51	Lists file statistics.
MODE Command on page 4-54	Changes command mode.
MXCI Command on page 4-55	Starts an MXCI session from the OSS environment.
OBEY Command on page 4-57	Executes MXCI commands and statements from a file. This file is referred to as an OBEY command file.
REPEAT Command on page 4-59	Re-executes a command. You can also use the exclamation point (!) command.
RESET PARAM Command on page 4-60	Clears the values of the specified parameter or all the parameters in the current session.
SET LIST_COUNT Command on page 4-62	Sets the maximum number of rows to be displayed for the next SELECT statement.
SET PARAM Command on page 4-63	Sets a value for a parameter used in queries to be executed.
SET SHOWSHAPE Command on page 4-66	Displays the access plans in effect. Generates the output of the SHOWSHAPE command for multiple SQL statements.
SET STATISTICS Command on page 4-69	Displays statistics automatically after each SQL statement.

For more information about utilities that you can run through MXCI, see Section 5, SQL/MX Utilities.

For more information on entering MXCI commands, see Entering a Command on page 1-3.

ADD DEFINE Command

Considerations for ADD DEFINE
Examples of ADD DEFINE

The ADD DEFINE command creates a new DEFINE in the current MXCI session. (ADD DEFINE is similar to the TACL command ADD DEFINE and the OSS shell command add_define.)

You can use defines only for SQL/MP objects.

You can use ADD DEFINE only within an MXCI session.

```
ADD DEFINE define
   [,CLASS MAP], FILE [\node.][[$volume.]subvol.]filename
```

define

is the name for the new DEFINE. To change an existing DEFINE, use the ALTER DEFINE command; *define* cannot be the same as an existing DEFINE name. See ALTER DEFINE Command on page 4-6.

A DEFINE name must begin with an equal sign (=) followed by a letter. The name is not case-sensitive and can be up to 24 characters, including alphanumeric characters and underscores (_).

A DEFINE name must not be a reserved word. Otherwise, you cannot select data using the DEFINE name of the table, view, or partition. See Appendix B, Reserved Words.

CLASS MAP

specifies that the DEFINE name *define* is associated with the name of a table, view, or partition. Use the DEFINE name in SQL statements as the logical name of an object, altering the DEFINE when you want to point to a different physical entity. MAP is the default CLASS.

FILE [*node*.][[$*volume*.]*subvol*.]*filename*

specifies the Guardian physical name of a table, view, or partition. *node* is the name of a node on a NonStop system, $*volume* is the name of a disk volume, *subvol* is the name of a subvolume, and *filename* is the name of a Guardian disk file. If the physical name is not fully qualified, it is expanded by using the current default node, volume, and subvolume.

The Guardian physical name must not include a reserved word. Otherwise, you cannot select data using the DEFINE name of the table, view, or partition. See Appendix B, Reserved Words.

If the Guardian physical name includes a reserved word, consider using the CREATE SQLMP ALIAS statement, instead of the ADD DEFINE command, to create a logical name mapping.

Considerations for ADD DEFINE

Scope of ADD DEFINE

A DEFINE stays in effect until you change or delete it or until you exit the current session.

A user can delete a DEFINE within MXCI. A user can delete a DEFINE within MXCI only that DEFINE was added using the MXCI ADD DEFINE or ALTER DEFINE command previously in the session. See the DELETE DEFINE Command on page 4-9.

Examples of ADD DEFINE

- Add a DEFINE that assigns the logical name =ORDERS to the table whose Guardian physical name is $SAMDB.SALES.ORDERS:

  ```
  ADD DEFINE =ORDERS, CLASS MAP, FILE $SAMDB.SALES.ORDERS;
  ```

  While this DEFINE is in effect, you can refer to the table as =ORDERS. The previous ADD DEFINE command is equivalent to:

  ```
  ADD DEFINE =ORDERS, FILE $SAMDB.SALES.ORDERS;
  ```

ALTER DEFINE Command

Considerations for ALTER DEFINE
Examples of ALTER DEFINE

The ALTER DEFINE command changes a DEFINE in the current MXCI session. (ALTER DEFINE is similar to the TACL command ALTER DEFINE.)

You can use ALTER DEFINE only within an MXCI session.

```
ALTER DEFINE define
   [,CLASS MAP], FILE [\node.][[$volume.]subvol.]filename
```

define

is the name of the existing DEFINE to change. DEFINEs can be inherited from the TACL process or the OSS shell and modified within MXCI. DEFINEs can also be created within MXCI and then modified within MXCI. See ADD DEFINE Command on page 4-4.

A DEFINE name must not be a reserved word. Otherwise, you cannot select data using the DEFINE name of the table, view, or partition. See Appendix B, Reserved Words.

CLASS MAP

specifies that the DEFINE name *define* is associated with the name of a table, view, or partition. Use the DEFINE name in SQL statements as the logical name of an object, altering the DEFINE when you want to point to a different physical entity. MAP is the default CLASS.

FILE [*node*.][[$*volume*.]*subvol*.]*filename*

specifies the Guardian physical name of a table, view, or partition. *node* is the name of a node on a NonStop system, $*volume* is the name of a disk volume, *subvol* is the name of a subvolume, and *filename* is the name of a Guardian disk file. If the physical name is not fully qualified, it is expanded by using the current default node, volume, and subvolume.

The Guardian physical name must not include a reserved word. Otherwise, you cannot select data by using the DEFINE name of the table, view, or partition. See Appendix B, Reserved Words.

Considerations for ALTER DEFINE

Scope of ALTER DEFINE

When you end an MXCI session, DEFINEs that you inherited from the TACL process or the OSS shell and modified within MXCI revert to the values they had when you started MXCI. You cannot alter an inherited DEFINE within MXCI.

Examples of ALTER DEFINE

- Alter a DEFINE that assigns the logical name =ORDERS to the table whose Guardian physical name is $MYVOL.SALES.ORDERS:

```
ALTER DEFINE =ORDERS, FILE $MYVOL.SALES.ORDERS;
```

CD Command

The CD command changes the current working session directory.

You can use CD only within an MXCI session.

```
CD [directory]
```

directory

is an absolute or relative path name that specifies the new current working directory. An absolute path name begins with a slash (/), the symbol representing the root directory. A relative path name defines a path relative to the current directory; it does not begin with /.

If you omit the name of the directory, the current working directory reverts to your home directory.

Considerations for CD

End of an MXCI Session

When your MXCI session ends, the working directory in effect when you started MXCI becomes the current working directory.

During an MXCI Session

During your MXCI session, you can use SHOW SESSION or its equivalent ENV to display the name of the current working directory.

Examples of CD

- Set the current working directory:

  ```
  cd /prjdir/testdir;
  ```

  The next sequence produces the same result but first resets the current directory to the root directory:

  ```
  cd /;
  cd prjdir/testdir;
  ```

- Set the current working directory to your home directory:

  ```
  cd;
  ```

DELETE DEFINE Command

The DELETE DEFINE command deletes a DEFINE in the current MXCI session. (DELETE DEFINE is similar to the TACL command DELETE DEFINE and the OSS shell command del_define.)

You can use DELETE DEFINE only within an MXCI session.

```
DELETE DEFINE define
```

define

is the name of the existing DEFINE to delete.

Considerations for DELETE DEFINE

Scope of DELETE DEFINE

After the DELETE DEFINE command executes, the specified DEFINEs inherited from the TACL process or the OSS shell are no longer in effect for the MXCI session, although these DEFINEs remain in effect for the TACL process or OSS shell.

The current user cannot delete inherited DEFINEs.

Examples of DELETE DEFINE

- Delete a DEFINE for the logical name =ORDERS:

  ```
  DELETE DEFINE =ORDERS;
  ```

DISPLAY USE OF Command

Considerations for DISPLAY USE OF
Examples of DISPLAY USE OF

The DISPLAY USE OF command provides usage information on statically compiled modules.

```
display use of [NOE] [module_dir 'module-directory-path']
  [ module 'module-name']  [ object 'object-name' ]
```

DISPLAY USE OF displays a list of modules and, for each module, a list of dependent objects. If you specify the `object 'object-name'` clause, DISPLAY USE OF displays a list of all dependent modules for that object.

Note. The current DISPLAY USE OF command does not support module names to be searched inside the Guardian location (path names starting with `/G/`). The new enhancement does not support the modules to be searched inside the Guardian location and displays the error as:

```
*** ERROR[19031] Searching modules in Guardian location is not
supported.
```

`NOE`

directs NonStop SQL/MX to scan modules only in the local node and not the nodes on the Expand network.

`module-directory-path`

is the path name of a directory, enclosed in single quotes.

`module-directory-path` is case-sensitive.

If you omit the `module_dir` clause, DISPLAY USE OF searches the default path, `/usr/tandem/sqlmx/USERMODULES`.

You can look for similar values by specifying only part of the characters of *`module-directory-path`* combined with “*” (asterisk) wild-card characters.

All of these specifications are valid:

```
[module_dir 'd1']
[module_dir './d1']
[module_dir '../d1']
[module_dir '/usr/tandem/sqlmx/USERMODULES/../d1']
[module_dir '*']
[module_dir '/*']
[module_dir '*/d1']
[module_dir '/*/d1']
[module_dir 'd1/*']
[module_dir '/d1/*']
```

module-name

is the name of the module to search for, enclosed in single quotes. NonStop SQL/MX searches all modules matching the pattern in `/usr/tandem/sqlmx/USERMODULES` or in the directory you specified with `module_dir`.

module-name is case-sensitive.

You can look for similar values by specifying only part of the characters of *module-name* combined with the “*” (asterisk) wild-card characters.

If you omit the module clause, DISPLAY USE OF will search for all modules.

All of these specifications are valid:

```
module 'CAT.SCH.M1'
module 'CAT.sch.*'
module 'CAT.*.*M'
```

object-name

is the fully qualified name of an object, such as a table or index name in SQL/MP alias name or Guardian format, enclosed in single quotes. *object-name* can be an SQL/MX object (tables, indexes, and SQL/MP aliases) or an SQL/MP object (tables and indexes.) The node is required for SQL/MP objects.

object-name is case-sensitive. Wild-card characters are not allowed.

These are valid specifications:

```
object 'TPCC.OE.CUSTOMER'
object '\SAMDBCAT.$PERSNL.EMPLOYEE.SALARY'
```

If you omit the `'object` *object-name*`'` clause, DISPLAY USE OF searches all objects (tables and indexes) in the matching modules.

Considerations for DISPLAY USE OF

Object Types

DISPLAY USE OF tracks dependencies among these types of objects when they are used with a static embedded SQL application written in either C, C++, COBOL, or Java:

- Compiled modules produced by the SQL/MX compiler
- SQL/MX objects: tables, indexes, and SQL/MP aliases
- SQL/MP objects: tables, and indexes

Information about the dependencies among these object types is derived from the Explain section of the compiled module.

If the module could not be loaded because the module name is not valid or the module name is valid but has been corrupted, NonStop SQL/MX displays an error. If the module could not be loaded because the name is valid but the module cannot be found, NonStop SQL/MX displays a warning.

Parallel Execution of DISPLAY USE OF

Suppose your system has many modules. When you run DISPLAY USE OF against all modules on the system, it might take 30 minutes or more.

If you name the modules with a distinguishing prefix, such as an indication of what application the module belongs to, you can use wild cards to run multiple instances of DISPLAY USE OF that each target a different set of modules, in different CPUS. Each of these instances will complete much more quickly. See the *SQL/MX Installation and Management Guide* for a detailed example of this.

Examples of DISPLAY USE OF

- Display all modules and dependent objects:

```
>>display use of;
Module: CAT.SCH.CONSTRAINTM
  Index: \NODE1.$DATA08.ORDERS.TI(CAT.SCH.T)
Object: CAT.SCH.T

Module: CAT.SCH.CURSOMEM
Object: \NODE1.$DATA08.ORDERS.T071CH
  Object: \NODE1.$DATA08.ORDERS.T071M
  Object: \NODE1.$DATA08.ORDERS.T071S
  Object: \NODE1.$DATA08.ORDERS.T071T
  Object: \NODE1.$DATA08.ORDERS.T071U

Module: CAT.SCH.TESTE001M
Table: \NODE1.$DATA08.ORDERS.RSWORKS
...
```

  (and so on for all modules).

- Display objects for all modules matching wild card “c*M”:

```
>>display use of module 'CAT.*.c*M';

Module: CAT.SCH.constraintM
Index: \NODE1.$DATA08.ORDERS.TI(CAT.SCH.T)
Table: CAT.SCH.T
Module: CAT.SCH.cursomeM
Table: \NODE1.$DATA08.ORDERS.T071CH
  Table: \NODE1.$DATA08.ORDERS.T071M
  Table: \NODE1.$DATA08.ORDERS.T071S
  Table: \NODE1.$DATA08.ORDERS.T071T
  Table: \NODE1.$DATA08.ORDERS.T071U
```

- Display use of a specified Guardian object in all modules:

```
>>display use of object '\NODE1.$DATA08.ORDERS.T1';

Object: \NODE1.$DATA08.ORDERS.T1
  Object: \NODE1.$DATA08.ORDERS.T1  Module:
TANDEM_SYSTEM_NSK.SCH.INS1M
Object: \NODE1.$DATA08.ORDERS.T1  Module:
TANDEM_SYSTEM_NSK.SCH.MULTCURM
Object: \NODE1.$DATA08.ORDERS.T1  Module:
TANDEM_SYSTEM_NSK.SCH.MULTIM
Object: \NODE1.$DATA08.ORDERS.T1  Module:
TANDEM_SYSTEM_NSK.SCH.STRUCTM Table: \NODE1.$DATA08.ORDERS.T1
Module: TANDEM_SYSTEM_NSK.SCH.T1INDM
Object: \NODE1.$DATA08.ORDERS.T1  Module:
TANDEM_SYSTEM_NSK.SCH.T1ROWSM
```

- Display use of a specified SQL/MP alias object in specified modules:

```
>>display use of module 'CAT.SCH.*' object 'CAT.SCH.T';

Object: CAT.SCH.T
  Table: CAT.SCH.T  Module: CAT.SCH.CONSTRAINTM
```

- Display modules that could not be loaded:

```
>>display use of object '\NODE1.$DATA08.ORDERS.T1';

Object: \NODE1.$DATA08.ORDERS.T11
  Object: \NODE1.$DATA08.ORDERS.T11  Module:
TANDEM_SYSTEM_NSK.SCH.INS1M
Modules not loaded:
  Module: SUPER.SUPER.MXOLTP
Error: -8809
  Module: SUPER.SUPER.NOWAITOLTM
Error: -8809
  Module: SUPER.SUPER.UPDATETESTM
Error: -8809
  Module: SUPER.SUPER.UPDATETESTNOWAITOLTM
Error: -8809
  Module: junk
Error: -8809
```

DISPLAY USE OF SOURCE

Examples of DISPLAY USE OF Source

The DISPLAY USE OF SOURCE Command displays the source SQL file (from which the module was created) for the given module. The syntax and utility of the new enhancement is as follows

```
DISPLAY USE OF [NOE] [MODULE_DIR 'module-directory-path']
SOURCE ['module-name']
```

`NOE`

directs NonStop SQL/MX to scan modules only in the local node and not the nodes on the Expand network.

`module-directory-path`

is the path name of a directory, enclosed in single quotes.

`module-directory-path` is case-sensitive.

If you omit the module_dir clause, DISPLAY USE OF SOURCE searches the default path, /usr/tandem/sqlmx/USERMODULES.

Note. The current DISPLAY USE OF command does not support module names to be searched inside the guardian location (that is, path names starting with /G/). Similarly, this command does not support the modules to be searched inside the guardian location.

You can search for similar values by specifying only part of the characters of module-directory-path combined with “*” (asterisk) wild-card characters.

All these specifications are valid:

```
[module_dir 'd1']
[module_dir './d1']
[module_dir '../d1']
[module_dir '/usr/tandem/sqlmx/USERMODULES/../d1']
[module_dir '*']
[module_dir '/*']
[module_dir '*/d1']
[module_dir '/*/d1']
[module_dir 'd1/*']
[module_dir '/d1/*']
```

`module-name`

is the name of the module to search for, enclosed in single quotes. NonStop SQL/MX searches all modules matching the pattern in /usr/tandem/sqlmx/USERMODULES or in the directory specified with module_dir.

`module-name` is case-sensitive.

You can search for similar values by specifying only part of the characters of module-name combined with the “*” (asterisk) wild-card characters.

If you omit the module clause, DISPLAY USE OF will search for all modules.

All of these specifications are valid:

```
module 'CAT.SCH.M1'
module 'CAT.sch.*'
module 'CAT.*.*M'
```

Note. If you do not specify the `module-name`, the command displays the source SQL file name for all the modules.

Examples of DISPLAY USE OF Source

- Display all modules and their corresponding source SQL files:

```
>>display use of SOURCE;


Module: CAT.SCH.CONSTRAINTM
Source Name: /E/NODE1/usr/user1/file1.sql

Module: CAT.SCH.CURSOMEM
Source Name: /E/NODE2/usr/user1/file2.sql

Module: CAT.SCH.TESTE001M
Source Name: /E/NODE1/usr/user2/file3.sql
```

- Display source SQL files for all modules matching wild card “c*M”:

```
>>display use of source 'CAT.*.c*M';

Module: CAT.SCH.constraintM
Source Name: /E/NODE11/usr/user2/constraint.sql

Module: CAT.SCH.cursomeM
Source Name: /E/NODE2/customer.sql

>>display use of module_dir '/usr/user1/all/modulestorage'
source 'CAT.ALL.MODULE1';

Module: CAT.ALL.MODULE1
Source Name: /E/NODE11/usr/user2/constraint.sql
```

DISPLAY USE OF ALL | INVALID MODULES

Considerations for DISPLAY USE OF ALL | INVALID MODULES

Examples of ALL | INVALID MODULES

If the INVALID keyword is specified in the command, it displays all the dependent modules along with their source SQL files for a given object that has become invalid because of the DDL changes to the object, after the module was created. If the ALL keyword is specified in the command, it displays all the dependent modules along with their source SQL files for a given object.

```
DISPLAY USE OF [NOE] [MODULE_DIR 'module-directory-path']
ALL|INVALID MODULES FOR 'object-name'
```

`NOE`

directs NonStop SQL/MX to scan modules only in the local node and not the nodes on the Expand network.

`module-directory-path`

is the path name of a directory, enclosed in single quotes.

`module-directory-path` is case-sensitive.

Note. The current DISPLAY USE OF command does not support module names to be searched inside the guardian location (that is, path names starting with /G/). Similarly, this command also does not support modules to be searched inside the guardian location.

You can look for similar values by specifying only part of the characters of module-name combined with the "*" (asterisk) wild-card characters.

All of these specifications are valid:

```
[module_dir 'd1']
[module_dir './d1']
[module_dir '../d1']
[module_dir '/usr/tandem/sqlmx/USERMODULES/../d1']
[module_dir '*']
[module_dir '/*']
[module_dir '*/d1']
[module_dir '/*/d1']
[module_dir 'd1/*']
[module_dir '/d1/*']
```

`object-name`

is the fully qualified name of an object, such as SQL/MX object (tables, indexes, and SQL/MP aliases).

Note. The current design does not support SQL/MP aliases to know about the DDL changes performed on the underlying SQL/MP object. Hence, this command can consider a module to be valid even if the underlying SQL/MP object has been changed after the creation of the module.

`object-name` is case-sensitive. Wild-card characters are not allowed.

A valid example is:

`TPCC.OE.CUSTOMER`

`ALL | INVALID`

If you specify the `INVALID` clause, only the modules that are invalid, because of DDL operations performed on the `object-name` (after the module was created), will be listed (along with their source SQL filename). However, if the `ALL` clause is specified, there will be no timestamp comparison and all the modules that are dependent on the `object-name` will be displayed along with their source SQL file name.

Considerations for DISPLAY USE OF ALL | INVALID MODULES

Object Types

`DISPLAY USE OF` tracks dependencies among objects when they are used with a static embedded SQL application written in either C, C++, COBOL, or Java:

- Compiled modules produced by the SQL/MX compiler
- SQL/MX objects: tables, indexes, and SQL/MP aliases

NonStop SQL/MX displays an error if a module cannot be loaded because the module name is not valid, or the module name is valid but has been corrupted. Also, if the module was compiled by an older compiler and does not contain the desired information, like the source SQL file name that is needed for these new commands, NonStop SQL/MX displays an error.
The error message in this case would be:

```
Version of these modules is incompatible for this command.
```

Note. All the modules that could not be read because of this error, are listed below the above-mentioned error message.

Examples of ALL | INVALID MODULES

- Display invalid modules along with their corresponding source SQL files for a given object:

```
>>display use of INVALID MODULES FOR ‘CAT.SCH.TABLE1’;

Object: CAT.SCH.TABLE1

Module: CAT.SCH.MODTABLE1
Source Name: /E/NODE11/modulelist/module1.sql

Module: CAT.SCH.MODTABLE2
Source Name: /E/NODE11/modulelist/module2.sql
```

- Display invalid modules along with their corresponding source SQL files for a given object when the search directory (where the modules need to be searched) is specified

```
>>display use of module_dir ‘/usr/user1/module_storage’
INVALID MODULES FOR ‘CAT.SCH.TABLE2’;

Object: CAT.SCH.TABLE2

Module: CAT.SCH.MOD1
Source Name: /E/NODE11/modules/mod1.sql
```

- Display all modules along with their corresponding source SQL files for a given object:

```
>>display use of ALL MODULES FOR ‘CAT.SCH.TABLE1;

Object: CAT.SCH.TABLE1

Module: CAT.SCH.MODTABLE1
Source Name: /E/NODE11/modulelist/module1.sql

Object: CAT.SCH.TABLE1

Module: CAT.SCH.MODTABLE3
Source Name: /E/NODE11/modulelist/module3.sql
```

- Display all modules along with their corresponding source SQL files for a given object when the search directory (module_dir) is specified:

```
>>display use of module_dir ‘/usr/user1/module_storage’ ALL
MODULES FOR ‘CAT.SCH.TABLE2’;

Object: CAT.SCH.TABLE2

Module: CAT.SCH.MOD123
Source Name: /E/NODE11/modules/mod123.sql
```

DISPLAY_QC Command

Considerations for DISPLAY_QC
Examples of DISPLAY_QC

The DISPLAY_QC command generates and displays selected data from the result of the QUERYCACHE function. For a description of the result table of the QUERYCACHE function, see the QUERYCACHE Function on page 8-134.

You can use DISPLAY_QC only within an MXCI session.

```
DISPLAY_QC
```

Considerations for DISPLAY_QC

Using QUERYCACHE and DISPLAY_QC

The DISPLAY_QC command provides a shortcut method of seeing the most commonly used columns of the QUERYCACHE function.

Purpose of the QUERYCACHE Function Result

The query plan cache automatically collects statistics regarding its use. When invoked, the QUERYCACHE table-valued function collects and returns these statistics in a single row table. The statistics are reinitialized when an `mxcmp` session is started and each `mxcmp` session maintains an independent set of statistics.

Result of the DISPLAY_QC Command

The DISPLAY_QC command displays these selected columns from the QUERYCACHE function:

Column Name	Type	Source column in QUERYCACHE Function
AVGSIZE	CHAR(8)	AVG_PLAN_SIZE
CURSIZE	CHAR(8)	CURRENT_SIZE
MAXSIZE	CHAR(8)	MAX_CACHE_SIZE
NPINNED	CHAR(8)	NUM_PINNED
NRECOM	CHAR(8)	NUM_RECOMPILES
NRETR	CHAR(8)	NUM_RETRIES
NCACHE	CHAR(8)	NUM_CACHEABLE_PARSING + NUM_CACHEABLE_BINDING
NHITS	CHAR(8)	NUM_CACHE_HITS_PARSING + NUM_CACHE_HITS_BINDING

Note that some of the fields can take on the value OVERFLOW if the cache statistics value is too large.

Examples of DISPLAY_QC

```
>>SET SCHEMA SAMDBCAT.PERSNL;

--- SQL operation complete.

>SELECT * FROM EMPLOYEE;

Employee/Number First Name       Last Name      Dept/Num  Job/Code  Salary
--------------  --------------  --------------  --------  --------  ---------
             1  ROGER           GREEN             9000       100    175500.00
            23  JERRY           HOWARD            1000       100    137000.10
            29  JANE            RAYMOND           3000       100    136000.00
.
.
.
--- 62 row(s) selected.

>>DISPLAY_QC;


AVGSIZE   CURSIZE   MAXSIZE   NPINNED   NRECOM    NRETR     NCACHE    NHITS

-------  --------  --------  --------  --------  --------  --------  --------


31        35        1024      0         0         0         1         0


--- SQL operation complete.
```

DISPLAY_QC_ENTRIES Command

Considerations for DISPLAY_QC_ENTRIES
Examples of DISPLAY_QC_ENTRIES

The DISPLAY_QC_ENTRIES command generates and displays selected data from the result of the QUERYCACHEENTRIES function. For a description of the result table of the QUERYCACHEENTRIES function, see the QUERYCACHEENTRIES Function on page 8-138.

You can use DISPLAY_QC_ENTRIES only within an MXCI session.

```
DISPLAY_QC_ENTRIES
```

Considerations for DISPLAY_QC_ENTRIES

Using QUERYCACHEENTRIES and DISPLAY_QC_ENTRIES

The DISPLAY_QC_ENTRIES command provides a shortcut display of the most commonly used columns of the QUERYCACHEENTRIES function.

Purpose of the QUERYCACHEENTRIES Function Result

The query plan cache automatically collects statistics on each entry of the cache. When invoked, the QUERYCACHEENTRIES table-valued function collects and returns these statistics in a table with one row for each entry of the cache. The statistics are reinitialized when an `mxcmp` session is started. Each `mxcmp` session maintains an independent set of statistics.

Result of the DISPLAY_QC_ENTRIES Command

The DISPLAY_QC_ENTRIES command displays these selected columns from the QUERYCACHEENTRIES function:

Column Name	**Type**	**Source column in QUERYCACHEENTRIES Function**
ROWID	CHAR(8)	ROW_ID
TEXT	CHAR(36)	TEXT
NUMHITS	CHAR(8)	NUM_HITS
PH	CHAR(1)	PHASE
COMPTIME	CHAR(8)	COMPILATION_TIME
AVGHITTIME	CHAR(8)	AVERAGE_HIT_TIME

Note that some of the fields can take on the value OVERFLOW if the cache statistics value is too large. In addition, the TEXT field can be truncated with only the first 36 characters displayed.

Examples of DISPLAY_QC_ENTRIES

```
>>SET SCHEMA SAMDBCAT.PERSNL;

--- SQL operation complete.

>SELECT * FROM EMPLOYEE;

Employee/Number First Name       Last Name      Dept/Num  Job/Code  Salary
--------------  --------------  --------------  --------  --------  ---------
             1  ROGER           GREEN             9000       100    175500.00
            23  JERRY           HOWARD            1000       100    137000.10
            29  JANE            RAYMOND           3000       100    136000.00
.
.
.
--- 62 row(s) selected.


>SELECT * FROM DEPT;

Dept/Num  Dept/Name     Mgr    Rpt/Dept  Location
--------  ------------  -----  --------  ------------------
    1000  FINANCE          23      9000  CHICAGO
    1500  PERSONNEL       213      1000  CHICAGO
    2000  INVENTORY        32      9000  LOS ANGELES
.
.
.
--- 12 row(s) selected.


>SELECT * FROM JOB;

Job/Code  Job Description
--------  ------------------

     100  MANAGER
     200  PRODUCTION SUPV
     250  ASSEMBLER
.
.
.
--- 10 row(s) selected.

>>DISPLAY_QC_ENTRIES;

ROWID     TEXT                               NUMHITS   PH  COMPTIME  AVGHTIME
-------  ---------------------------------  --------  --  --------  --------

0         select * from job;                     0     B   88        0
1         select * from dept;                    0     B   115       0
2         select * from employee;                0     B   1605      0

--- SQL operation complete.
```

DISPLAY STATISTICS Command

Considerations for DISPLAY STATISTICS
Examples of DISPLAY STATISTICS

DISPLAY STATISTICS displays statistics about the last DML or PREPARE statement executed within the current MXCI session.

Use DISPLAY STATISTICS only within an MXCI session.

```
DISPLAY STATISTICS
```

Considerations for DISPLAY STATISTICS

When you issue the DISPLAY STATISTICS command, MXCI displays:

Start time	Time when the query is first issued from MXCI.
End time	Time when the query ends and results are displayed.
Elapsed time	Equals the sum of the compile time and execution time.
Compile time	Amount of time to prepare the query.
Execution time	Amount of time used by the SQL executor to execute the query.
Number of records accessed and used	Records accessed gives a count of the number of records accessed in each table. This count includes records examined by the disk process, the file system, and the SQL executor. Records used gives a count of records actually used by the statement. For INSERT and FETCH operations, the count is always 0 or 1. For UPDATE, DELETE, and SELECT operations, the count can be greater than 1.
Disk I/Os	Disk I/Os gives a count of the number of disk reads caused by accessing this table.
Message count	Message count gives a count of the number of messages sent to execute operations on this table. For example, a FETCH operation through a secondary index generally sends two messages.
Message bytes	Message bytes gives a count of the message bytes sent to access this table.
Lock	Lock displays flags indicating that lock waits occurred (W) or that lock escalations occurred (E) for the table. If this field is blank, no locks were obtained during the processing of this statement.

Examples of DISPLAY STATISTICS

- Suppose that this was the last DML command issued:

```
DELETE FROM invent.partsupp
WHERE suppnum NOT IN
  (SELECT suppnum FROM supplier
   WHERE state='TEXAS');
```

```
--- 41 row(s) deleted.
```

You can display statistics for this statement by using the DISPLAY STATISTICS command:

```
DISPLAY STATISTICS;


Start Time                2001/08/22 09:24:50.188
End Time                  2001/08/22 09:24:51.966
Elapsed Time                         00:00:01.777
Compile Time                         00:00:00.633
Execution Time                       00:00:01.145

Table Name            Records      Records    Disk    Message       Message    Lock
                     Accessed         Used    I/Os      Count         Bytes

SAMDBCAT.INVENT.PARTSUPP
                           49           49       0          2          9056       0
SAMDBCAT.INVENT.SUPPLIER
                           49            8       0         11         31304       0
SAMDBCAT.INVENT.PARTSUPP
                           41           41       0          6         17144       0
"\MYSYS.$SAMDB".INVENT.XSUPORD
                           41           41       2          9         23056       0
>>log;
```

ENV Command

ENV displays attributes of the current MXCI session. You can use ENV (or SHOW SESSION Command) only within an MXCI session.

```
ENV
```

ENV displays these attributes:

CURRENT DIRECTORY	Path name of the current working directory. You can change it with the CD command.
HOME DIRECTORY	Default directory.
LIST_COUNT	Current list count.
LOG FILE	Current log file.
MESSAGEFILE	Current message file.
TERMINAL CHARSET	Current character set
MESSAGEFILE LANG	Language of the text in the message file.
MESSAGEFILE VRSN	Version of the message file; its value is stored in the message.
SQL CATALOG	Default catalog.
SQL SCHEMA	Default schema.
TRANSACTION ID	Transaction identifier of the current transaction if one is in progress.
TRANSACTION STATE	Transaction status (in progress or not in progress).
WARNINGS	Current state of warnings (on or off).

Examples of ENV

- An ENV command and its output:

```
>>env;
----------------------------------
Current Environment
----------------------------------
CURRENT DIRECTORY  /usr/manager/bin
HOME DIRECTORY     /usr/manager
LIST_COUNT         4294967295
LOG FILE
MESSAGEFILE        /usr/manager/bin/mxcierrors.cat
TERMINAL CHARSET   ISO88591
MESSAGEFILE LANG   US English
MESSAGEFILE VRSN   {2003-12-11 13:56 NSK:SQUAW/SUPER.SUPER}
SQL CATALOG        CAT
SQL SCHEMA         SCH
TRANSACTION ID
TRANSACTION STATE  not in progress
WARNINGS           on
>>
```

- An ENV command showing the effect of the NSK NAMETYPE:

```
>>set nametype NSK;
>>env;
----------------------------------
Current Environment
----------------------------------
CURRENT DIRECTORY  /usr/manager/bin
HOME DIRECTORY     /usr/manager
LIST_COUNT         4294967295
LOG FILE
MESSAGEFILE        /usr/manager/bin/mxcierrors.cat
TERMINAL CHARSET   ISO88591
MESSAGEFILE LANG   US English
MESSAGEFILE VRSN   {2003-12-11 13:56 NSK:SQUAW/SUPER.SUPER}
SQL CATALOG        CAT (\SQUAW.$SYSTEM)
SQL SCHEMA         SCH (ZSDGTXNC)
TRANSACTION ID
TRANSACTION STATE  not in progress
WARNINGS           on
>>
```

ERROR Command

The ERROR command displays the error text associated with an error number.

You can use ERROR only within an MXCI session.

```
ERROR number [,BRIEF]
```

number

is an unsigned integer that identifies the error you want described.

BRIEF

displays the error text associated with the specified *number*. The ERROR command returns the same information with or without this keyword.

Examples of ERROR

- Display the text of error 1000:

```
ERROR 1000;
```

```
*** SQLSTATE (Err): 42000 SQLSTATE (Warn): 01500
*** ERROR[1000] A syntax error occurred.
```

Exclamation Point (!) Command

The exclamation point (!) command re-executes a previous MXCI command.

You can use `!` only within an MXCI session.

```
! [text | [-]number]
```

text

specifies the text of the most recent use of a command. The command must have been executed beginning with *text*, but *text* need be only as many characters as necessary to identify the command. Leading blanks are ignored.

[-]*number*

is an integer that identifies a command in the history buffer. If *number* is negative, it indicates the position of the command in the history buffer relative to the current command; if *number* is positive, it is the ordinal number of a command in the history buffer.

The HISTORY command displays the commands or statements in the history buffer. See HISTORY Command on page 4-44.

To re-execute the immediately preceding command, enter an exclamation point without specifying text or number. If you enter more than one MXCI command on a line, the exclamation point re-executes only the last command on the line.

Examples of !

- Suppose that you have a series of statements you have executed. Re-execute the last SELECT:

```
>>! SELECT;
>>SELECT * FROM samdbcat.invent.partsupp;

PARTNUM  SUPPNUM  PARTCOST      QTY_RECEIVED
-------  -------  ------------  ------------
   2000       95       1000.00            10
   2010       99         30.00            20
   2020      186        200.00            30
   ...
```

- Re-execute the second to the last command:

```
!-2;
```

- Re-execute the second command in the history buffer:

```
! 2;
```

EXIT Command

The EXIT command ends an MXCI session and returns control to the process from which you started MXCI.

You can use EXIT only within an MXCI session.

```
EXIT
```

Considerations for EXIT

Effect of EXIT on Active Transactions

A transaction can be user-initiated or system-initiated. If you attempt to end an MXCI session when either type of transaction is active, MXCI prompts you to specify whether to commit or roll back the work of the transaction.

Examples of EXIT

- End an MXCI session:

  ```
  >>EXIT;
  ```

  ```
  End of MXCI Session
  ```

FC Command

Examples of FC

The FC command allows you to edit and reissue an MXCI command in the history buffer. You can display the commands in the history buffer by using the HISTORY command. For more information about the history buffer, see HISTORY Command on page 4-44.

You can use FC only within an MXCI session.

```
FC [text | [-]number]
```

text

is the beginning text of a command in the MXCI history buffer. Case is not significant in matching the text to a command.

[-]*number*

is either a positive integer that is the ordinal number of a command in the MXCI history buffer or a negative integer that indicates the position of a command relative to the most recent command.

Without *text* or *number* FC retrieves the most recent command.

A semicolon (;) is not required after the FC command.

As each line is displayed, you can modify it by entering these commands (in uppercase or lowercase letters) on the line below the displayed command:

D	Deletes the character immediately above the letter D. Repeat to delete more characters.
I*characters*	Inserts *characters* in front of the character immediately above the letter I.
R*characters*	Replaces existing characters one-for-one with *characters*, beginning with the character immediately above the letter R.
characters	Replaces existing characters one-for-one with *characters*, beginning with the first character immediately above *characters*. *characters* must begin with a nonblank character.

To specify more than one editing command on a line, separate the editing commands with a double slash (//).

The end of a line terminates a command. After you edit the last line of the command, MXCI displays the command again and allows you to edit it again. To stop editing and execute the edited command, press Return without entering any editing commands.

To terminate a command without saving changes to the command, use the double slash (//), and then press Return.

Examples of FC

- Re-execute the most recent command that begins with SH:

```
>>FC SH;
>>SHOW SESSION;
..
```

Pressing Return executes the SHOW SESSION command.

- Correct a statement entered incorrectly:

```
>>SELECR * FROM MYTABLE;

*** ERROR[15001] A syntax error at or before:
selecr * from $boy000.persnl.employee;
       ^

>>FC;
>>SELECR * FROM MYTABLE;
..     T
>>SELECT * FROM MYTABLE;
..
```

Pressing Return executes the corrected SELECT statement.

- Modify a previously executed statement:

```
>>SELECT SUPPNAME, CITY, STATE
>+FROM INVENT.SUPPLIER
>+WHERE SUPPNUM = 4;

-- 0 row(s) selected.
>>FC;
>>SELECT SUPPNAME, CITY, STATE
..

>>FROM INVENT.SUPPLIER
..
>>WHERE SUPPNUM = 4;
..DDDDDDDDDDDDDDDDD
>>;
..
```

Pressing Return lists all of the suppliers.

GTACL Command

Considerations for GTACL

Examples of GTACL

The GTACL command allows you to run the TACL commands, such as, FUP, PSTATE, STATUS, TACL macros, programs, and utilities from the MXCI interface.

Note. The GTACL command is available only on systems running J06.09 and later J-series RVUs and H06.20 and later H-series RVUs. This command is introduced to inherit the same property as provided in the `gtacl` command of the Open System Services (OSS) environment.

In the MXCI session, the GTACL command is not case-sensitive; it can be invoked using GTACL or gtacl.

```
GTACL [option ...] [operands]
```

`option`

All filename and pathname arguments used with GTACL options must be specified using the OSS pathname syntax.

`operands`

Operands used with the GTACL command must follow the GTACL option specifications. The operands can be any TACL command.

For more information about the *`option`* and *`operands`*, run the following command at the MXCI prompt:

```
SH man gtacl;
```

For information about the `gtacl` command in the OSS environment, see the `gtacl(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*.

Considerations for GTACL

The GTACL command writes the data to the standard output file. If you issue the LOG command in the current MXCI session, the GTACL command still writes the data to the standard output file, only the command will be logged.

Examples of GTACL

- To identify the users who have currently logged in, run the following command:

  ```
  >>gtacl -c "who";
  ```

The output is displayed as:

```
gtacl[9]: warning: unable to propagate all environment
variables
Home terminal: $ZTN0.#PTMMMM5
TACL process: \NSAA12.$Z1QA
Primary CPU: 2 (NSE-D)
Default Segment File: $DATA02.#0000085
  Pages allocated: 56  Pages Maximum: 1036
  Bytes Used: 99116 (4%)  Bytes Maximum: 2121728
Current volume: $DATA02.LAKDGTST
Saved volume:   $SYSTEM.SYSTEM
Userid: 255,255  Username: SUPER.SUPER  Security: "NUNU"
Logon name: SUPER.SUPER
```

- To invoke a TACL executable file, run the following command:

```
>>gtacl
```

The output is displayed as:

```
gtacl[9]: warning: unable to propagate all environment
variables
```

```
TACL 1>
```

The TACL EXIT command stops the TACL command and returns to the MXCI session.

GET NAMES OF RELATED NODES Command

The GET NAMES OF RELATED NODES command displays the names of the transitive closure of nodes that are related to the specified node. A node is related to another node if they have one or more catalogs in common. The specified node is included in the output.

```
GET NAMES OF RELATED NODES [FOR node];
```

node

is the node name for which the list of related nodes is requested. The default is the local node.

Error Conditions for GET NAMES OF RELATED NODES

An error for the GET NAMES OF RELATED NODES command occurs when:

- The specified node does not exist.
- The specified node cannot be accessed.
- One or more required nodes other than the specified node cannot be accessed.
- SQL/MX is not installed on the specified node.
- An invalid node name is specified.

Example of GET NAMES OF RELATED NODES

In this example, catalog `CAT1` is visible on `\LONDON` and `\GLASGOW`, catalog `CAT2` is visible on `\LONDON` and `\CPH`, and catalog `CAT3` is visible on `\BERLIN` and `\CPH`:

```
>> GET NAMES OF RELATED NODES FOR \LONDON;
```

```
NODES:\BERLIN
      \CPH
      \GLASGOW
      \LONDON
```

GET NAMES OF RELATED SCHEMAS Command

The GET NAMES OF RELATED SCHEMAS command displays the names of the transitive closure of schemas related to the specified schema. Schemas are related if a view, a trigger, or a constraint in one schema references an object in the other schema. The specified schema is included in the output.

```
GET NAMES OF RELATED SCHEMAS FOR schema-name;
```

schema-name

is the ANSI name of the schema for which the list of related schemas is requested. There is no default for the *schema-name*. However, the MXCI default catalog applies.

Error Conditions for GET NAMES OF RELATED SCHEMAS

An error for the GET NAMES OF RELATED SCHEMAS command occurs when:

- The catalog of the specified schema is not visible on the local node.
- The specified schema does not exist.
- The specified schema has no metadata on the local node and the remote node(s), where an automatic reference for the schema's catalog exists, cannot be accessed.
- A related schema has no metadata on the local node and the remote node(s), where an automatic reference for the schema's catalog exists, cannot be accessed.
- An invalid schema name is specified.

Example of GET NAMES OF RELATED SCHEMAS

In this example, view `CAT.SCH1.V1` references tables `CAT.SCH2.T2` and `CAT.SCH3.T3`. A referential integrity constraint exists between tables `CAT.SCH3.TX` and `CAT.SCH4.TY`.

```
>> GET NAMES OF RELATED SCHEMAS FOR CAT.SCH1;

SCHEMAS:CAT.SCH1
        CAT.SCH2
        CAT.SCH3
        CAT.SCH4
```

GET NAMES OF RELATED CATALOGS

The GET NAMES OF RELATED CATALOGS command displays the names of the transitive closure of catalogs related to the specified catalog. Catalogs are related if a view, a trigger, or a constraint in one catalog references an object in the other catalog. The specified catalog is included in the output.

```
GET NAMES OF RELATED CATALOGS FOR catalog-name;
```

catalog-name

is the ANSI name of the catalog for which the list of related catalogs is desired. There is no default for the *catalog-name*.

Error Conditions for GET NAMES OF RELATED CATALOGS

An error for the GET NAMES OF RELATED CATALOGS command occurs when:

- The specified catalog is not visible on the local node.
- The specified catalog has a manual reference on the local node, and the remote node(s), where an automatic reference for the catalog exists, cannot be accessed.
- A related catalog has a manual reference on the local node, and the remote node(s), where an automatic reference for the catalog exists, cannot be accessed.
- An invalid catalog name is specified.

Example of GET NAMES OF RELATED CATALOGS

In this example, view CAT1.SCH.V1 references tables CAT2.SCH.T2 and CAT3.SCHA.T3. A referential integrity constraint exists between tables CAT3.SCHB.TX and CAT4.SCH.TY.

```
>> GET NAMES OF RELATED CATALOGS FOR CAT1;

CATALOGS:CAT1
         CAT2
         CAT3
         CAT4
```

GET VERSION OF SYSTEM

The GET VERSION OF SYSTEM command displays the SQL/MX Software Version (MXV) information of a specified node. If no node is specified, the MXV of the local node is returned.

```
GET VERSION OF SYSTEM [ node ];
```

node

is the node name for which the SQL/MX Software Version information is requested. The default is the local node.

Error Conditions for GET VERSION OF SYSTEM

An errors for the GET VERSION OF SYSTEM command occurs when:

- The specified node does not exist.
- The specified node cannot be accessed.
- SQL/MX is not installed on the specified node.
- An invalid node name is specified.

Example of GET VERSION OF SYSTEM

```
>> GET VERSION OF SYSTEM \NODEX;

VERSION: 1200
```

GET VERSION OF SCHEMA Command

The GET VERSION OF SCHEMA command displays the schema version of the specified schema.

```
GET VERSION OF SCHEMA schema-name;
```

schema-name

is the ANSI name of the schema for which the schema version is requested. There is no default for the *schema-name.* However, the MXCI default catalog applies.

Error Conditions for GET VERSION OF SCHEMA

An error for the GET VERSION OF SCHEMA command occurs when:

- The catalog of the specified schema does not exist.
- The specified schema does not exist.
- The specified schema is not defined on the local node and the remote node(s), where a full replica of the schema's catalog exists, cannot be accessed.
- An invalid schema name is specified.

Examples of GET VERSION OF SCHEMA

```
>> GET VERSION OF SCHEMA MYCAT.MYSCH;

VERSION: 1200


>> GET VERSION OF SCHEMA YOURDB;

VERSION: 3000
```

GET VERSION OF SYSTEM SCHEMA Command

The GET VERSION OF SYSTEM SCHEMA command displays the system schema version of the specified node. If no node is specified, the system schema version of the local node is displayed.

```
GET VERSION OF SYSTEM SCHEMA [ ON  node ];
```

node

is the node name for which the system schema version information is requested. The default is the local node.

Error Conditions for GET VERSION OF SYSTEM SCHEMA

An error for the GET VERSION OF SYSTEM SCHEMA command occurs when:

- The specified node does not exist.
- The specified node cannot be accessed.
- SQL/MX is not installed on the specified node.
- SQL/MX has not been initialized on the specified node.
- An invalid node name is specified.

Example of GET VERSION OF SYSTEM SCHEMA

```
>> GET VERSION OF SYSTEM SCHEMA ON \NODEY;

VERSION: 1000
```

GET VERSION OF Object Command

The GET VERSION OF Object command displays the object schema version (OSV) and the object feature version (OFV) of the specified database object.

```
GET VERSION OF object-type object-name;
```

object-type

is the type of object for which version information is requested. The following are the object types:

- TABLE
- INDEX
- VIEW
- CONSTRAINT
- TRIGGER
- STORED PROCEDURE
- MPALIAS

  There is no default object type.

object-name

is the ANSI name of the object, of the specified type, for which the version information is requested. There is no default for *object-name*. However, MXCI default catalog and schema apply.

Error Conditions for GET VERSION OF Object

An error for the GET VERSION OF Object command occurs when:

- The object of the specified catalog does not exist.
- The schema of the specified object schema does not exist.
- The specified object is not defined on the local node and the remote node(s), where a full reference of the object's catalog exists, cannot be accessed.
- An invalid object name is specified.

Example of GET VERSION OF Object

```
>> GET VERSION OF TABLE PROD.ORDERDB.ORDER_DETAIL;

OBJECT SCHEMA VERSION:  3000

OBJECT FEATURE VERSION: 1200
```

GET VERSION OF MODULE Command

The GET VERSION OF MODULE command displays the version of the specified module.

```
GET VERSION OF MODULE 'module-name';
```

module-name

is the name of the module for which the module version information is requested. There is no default for *module-name*, and MXCI default catalog and schema do not apply. If the module exists in an OSS directory other than USERMODULES, the full OSS path must be specified as the module name.

Note. The module name must be specified within single quotes.

Error Conditions for GET VERSION OF MODULE

An error for the GET VERSION OF MODULE command occurs when:

- The specified module does not exist.
- The specified module cannot be accessed.
- An invalid module name is specified.

Example of GET VERSION OF MODULE

```
>> GET VERSION OF MODULE 'PROD.CUSTDB.CUSTOMER_MAINTENANCE';

VERSION:  1200
```

GET VERSION OF PROCEDURE Command

The GET VERSION OF PROCEDURE command displays the plan version of the specified procedure in the specified module.

```
GET VERSION OF PROCEDURE ('module-name','procedure-name');
```

module-name

is the name of the module for which version information is desired. There is no default for *module-name* and MXCI default catalog and schema do not apply. If a module exists in metadata and a module with the same name also exists in the USERMODULES OSS directory, the module in metadata is reported. If the module exists in an OSS directory other than USERMODULES, the full OSS path must be specified as module name.

Note. The module name must be specified in single quotes.

procedure-name

is the name of the specific procedure in the module for which plan version information is requested.

Error Conditions for GET VERSION OF PROCEDURE

An error for the GET VERSION OF PROCEDURE occurs when:

- The specified module does not exist.
- The specified procedure name does not exist in the specified module.
- The specified procedure name is invalid.
- The specified module cannot be accessed.
- An invalid module name is specified.

Example of GET VERSION OF PROCEDURE

```
>> GET VERSION OF PROCEDURE ('/usr/sqlmods/CAT.SCH.MOD',
'PROC47');

VERSION:  1200
```

GET VERSION OF STATEMENT Command

The GET VERSION OF STATEMENT command displays the plan version of the specified prepared statement.

```
GET VERSION OF STATEMENT statement-name;
```

statement-name

is the name of a prepared statement for which plan version information is requested. There is no default for *statement-name*. The statement must have been previously prepared in the same MXCI session as the one where the GET VERSION command is issued.

Error Conditions for GET VERSION OF STATEMENT

An error for the GET VERSION OF STATEMENT command occurs when:

- The specified statement does not exist.
- An invalid statement name is specified.

Example of GET VERSION OF STATEMENT

```
>> prepare myquery from select * from cat.sch.t22;

>> GET VERSION OF STATEMENT myquery;

VERSION:  1200
```

HISTORY Command

The HISTORY command displays recently executed MXCI commands, identifying each command by a number you can use to re-execute or edit the command with FC. See FC Command on page 4-30.

You can use HISTORY only within an MXCI session.

```
HISTORY [number]
```

number

is the number of commands to display. The default number is 10.

You can use the FC command to edit and re-execute a command in the history buffer, or use the exclamation point command (!) to re-execute a command without modifying it.

Examples of HISTORY

- Display the three most recent MXCI commands and use FC to redisplay one:

```
>>HISTORY 3;

1> SHOW SESSION;
2> SELECT * FROM PERSNL.DEPT;
3> HISTORY 3;
>>FC 2;
>>SELECT * FROM PERSNL.DEPT;
..
```

Now you can use the edit capabilities of FC to modify and execute a different SELECT statement.

INFO DEFINE Command

INFO DEFINE displays the logical and physical names of DEFINEs in the current MXCI session. (INFO DEFINE is similar to the TACL command INFO DEFINE and the OSS shell command info_define.)

```
INFO DEFINE [ALL]
```

`ALL`

displays information about all the DEFINEs. If you do not specify ALL, the INFO DEFINE command displays information about the current class MAP DEFINEs, which are associated with the names of tables, views, or partitions.

Examples of INFO DEFINE

- Display information about the current class MAP DEFINEs:

```
>> INFO DEFINE;

=ORDERS
$SAMDB.SALES.ORDERS
=CUSTOMER
$SAMDB.SALES.CUSTOMER
=ODETAIL
$SAMDB.SALES.ODETAIL
=PARTS
$SAMDB.SALES.PARTS
```

- Display information about all the DEFINES:

```
>> INFO DEFINE ALL;

=ORDERS, class MAP, FILE \MYSYS\$SAMDB.SALES.ORDERS
=CUSTOMER, class MAP, FILE \MYSYS\$SAMDB.SALES.CUSTOMER
=ODETAIL, class MAP, FILE \MYSYS\$SAMDB.SALES.ODETAIL
=PARTS, class MAP, FILE \MYSYS\$SAMDB.SALES.PARTS
=_DEFAULTS, class DEFAULTS, VOLUME $SYSTEM.NOSUBVOL
```

INVOKE Command

Examples of INVOKE

The INVOKE command generates a record description that corresponds to a row in the specified table or view. The record description includes a data item for each column in the table or view except the SYSKEY column; it includes the SYSKEY column of a view only if the view explicitly listed the column in its definition.

You can use this version of INVOKE only within an MXCI session.

```
INVOKE table
```

table

names the table or view for which to generate a record description.

Examples of INVOKE

- Generate a record description of table EMPLOYEE:

```
INVOKE EMPLOYEE;
-- Definition of table EMPLOYEE
-- Definition current  Mon Apr 24 16:03:04 2000
  (
    EMPNUM        NUMERIC(4, 0) UNSIGNED NO DEFAULT
      HEADING 'Employee/Number' NOT NULL NOT DROPPABLE
  , FIRST_NAME    CHAR(15) CHARACTER SET ISO88591 COLLATE
      DEFAULT DEFAULT ' '
      HEADING 'First Name' NOT NULL NOT DROPPABLE
  , LAST_NAME     CHAR(20) CHARACTER SET ISO88591 COLLATE
      DEFAULT DEFAULT ' '
      HEADING 'Last Name' NOT NULL NOT DROPPABLE
  , DEPTNUM       NUMERIC(4, 0) UNSIGNED NO DEFAULT
      HEADING 'Dept/Num' NOT NULL NOT DROPPABLE
  , JOBCODE       NUMERIC(4, 0) UNSIGNED DEFAULT NULL
      HEADING 'Job/Code'
  , SALARY        NUMERIC(8, 2) UNSIGNED DEFAULT NULL
      HEADING 'Salary'
  )

--- SQL operation complete.
```

LOG Command

[Considerations for LOG](#)
[Examples of LOG](#)

The LOG command starts or stops MXCI logging to a disk file. When logging is in effect, MXCI writes the commands you enter to a file (in addition to executing them) and writes the output of the commands to the file.

Use LOG only within an MXCI session.

```
LOG [log_file {[COMMAND[S]] | [RESULT [ONLY] ]} [CLEAR]]
```

log_file

is the name of a file to which MXCI writes the commands you use and the command output. LOG closes the previous log file, if any, and opens *log_file* as the new log file. The path name can be either an absolute path name or a relative path name.

To stop logging, omit *log_file*.

△ **Caution.** To ensure that log information is retained for an MXCI session, use a unique name for the log file. For more information, see Concurrent MXCI Sessions on page 4-48.

[COMMAND[S]]

logs only MXCI input, not output or prompts.

[RESULT]

logs the output of a command or query, all success and error messages, and row count information that are entered from an MXCI session. It does not log the entered commands or queries.

[RESULT ONLY]

logs only the output of a command or query from an MXCI session. The ONLY option must be used only with the RESULT option. If it is used with the COMMAND[S] option, a syntax error is displayed.

Note. The RESULT and ONLY options are available only on systems running J06.09 and later J-series RVUs and H06.20 and later H-series RVUs. The ONLY option does not display the rows selected, operation complete status, and the result headings. The ONLY option writes the statistics, control, and explain outputs to the *log_file*.

CLEAR

clears *log_file* before logging. If you omit CLEAR, LOG appends the new log to existing data in *log_file*.

Considerations for LOG

Contents of the Log File

The log file includes all lines you enter except FC editing lines, including the final version of any line you edit by using FC. It also includes the prompts for lines that you enter and all text that MXCI displays or prints in response to those lines, including output from commands and diagnostic messages—except for output from the CD, FC, HISTORY, LS, SH, GTACL, and ! commands.

Concurrent MXCI Sessions

If two or more concurrent MXCI sessions use the same *log_file* name in a LOG command, each MXCI session writes information to the same log file. After the log file is closed, you cannot determine which information was written by each MXCI session. To ensure that log information is retained for a session, use a unique name for each log file.

Examples of LOG

- Start logging only commands to an OSS text file in the current directory, clearing the file first:

```
>>LOG myfile COMMANDS CLEAR;
```

- Stop logging:

```
>>LOG;
```

- Start logging with CLEAR option:

```
>>LOG myfile CLEAR;
>>select * from mytable;
>>select * from tab;
>>LOG;
```

The log file displays the following information:

```
>>select * from mytable;

  I
  -----------
          1
          2
          3
          4
          5

  --- 5 row(s) selected.
>>select * from tab;

*** ERROR[4082] Table, view or stored procedure CAT.SCH.TAB
does not exist or is inaccessible.
```

```
*** ERROR[8822] The statement was not prepared.

>>LOG;
```

- Start logging only commands:

```
>>LOG myfile COMMANDS CLEAR;
>>select * from mytable;
>>select * from tab;
>>LOG;
```

The log file displays the following information:

```
>>select * from mytable;
>>select * from tab;
>>LOG;
```

- Start logging using RESULT option:

```
>>LOG myfile RESULT CLEAR;
>>select * from mytable;
>>select * from tab;
>>LOG;
```

The log file displays the following information:

```
  I
  -----------
          1
          2
          3
          4
          5

   --- 5 row(s) selected.

*** ERROR[4082] Table, view or stored procedure CAT.SCH.TAB
does not exist or is inaccessible.

*** ERROR[8822] The statement was not prepared.
```

- Start logging using `RESULT` and `ONLY` options:

```
>>LOG myfile RESULT ONLY CLEAR;
>>select * from mytable;
>>select * from tab;
>>LOG;
```

The log file displays the following information:

```
          1
          2
          3
          4
          5
```

LS Command

[Considerations for LS](#)
[Examples of LS](#)

LS lists file statistics.

You can use LS only within an MXCI session.

```
LS [-abcCdfFgilmnopqrRstux1] [file | directory]...
```

`-abcCdfFgilmnopqrRstux1`

indicates some of the standard flags available to you through your platform's shell **ls** command, as follows:

`-a`	Lists all entries in directory, including those beginning with dot (.)
`-b`	Displays nonprintable characters in octal notation.
`-c`	Uses the time of last property change, mode change, and so on, for sorting (when used with -t option) or for displaying (when used with -l, -g, -n, -o, or -u options).
`-C`	Sorts output vertically in a multicolumn format, the default.
`-d`	Displays only the information for the directory that is named, rather than for its contents. This option is useful with the -l option to get the status of a directory.
`-f`	This option turns off the -l, -t, -s, and -r options and turns on the -a option; the option uses the order in which entries appear in the directory.
`-F`	Puts a / (slash) after each file name if the file is a directory and an * (asterisk) after each file name if the file can be executed.
`-g`	Displays the same information as the -l option, except for the owner, which is not displayed.
`-i`	Displays the inode number in the first column of the report for each file.
`-l`	Displays the mode, number of links, owner, group, size, time of last modification for each file, and file name.
`-m`	Uses stream output format (a comma-separated series).
`-n`	Displays the same information as the -l option, except that it displays user and group IDs instead of user and group names.
`-o`	Displays the same information as the -l option, except for the group, which is not displayed. The -n option overrides the -o option.
`-p`	Puts a slash after each file name if that file is a directory.
`-q`	Displays nonprintable characters in file names as a ? (question mark) character if output is sent to the monitor (the default destination).
`-r`	Reverses the order of the sort, giving reverse collation, or the oldest first, as appropriate.

-R Lists all subdirectories recursively.

-s Gives space used in 512-byte units (including indirect blocks) for each entry.

-t Sorts by time of last modification (latest first) instead of by name, before sorting the operands by the collating sequence.

-u Uses the time of the last access instead of the time of the last modification for sorting (when used with the -t option) or for displaying (when used with the -l option). The -u option has no effect unless used with either the -t or -l option or both.

-x Sorts output horizontally in a multicolumn format.

-1 Forces an output format of one entry per line: this is the default format when output is not directed to a monitor.

To display all the command-line arguments that your shell supports, at the MXCI prompt, type:

```
sh man ls;
```

A reference page of the LS command explains the command-line arguments that your shell supports. At the end of the reference page, the MXCI prompt automatically appears.

Considerations for LS

Output

The LS command writes to the standard output file the names of the specified files, along with any other information you ask for by specifying options. If logging is enabled—that is, you have issued the LOG command in the current session—the LS command still writes to the standard output file, not to the log file.

Defaults

If you do not specify a file or directory, LS displays the files in the current directory. By default, LS displays information by file name.

Examples of LS

- Change to the `/usr/jbrook` directory and display the files in the directory:

```
>>cd /usr/jbrook;
>>ls;
logjb         myfile        sh_history
```

- Display detailed information about the files named `logjb` and `myfile` in the `/usr/jbrook` directory:

```
>>ls -l logjb myfile;
-rw-rw-rw-   1 PUBS.JBROOK        PUBS        4856 Mar  6
```

```
13:03 logjb
-rw-rw-rw-   1 PUBS.JBROOK        PUBS       18961 Feb  8
14:11 myfile
```

- Display detailed information about the directory named `/usr`:

```
>>ls -d -l /usr;
drwxrwxr-x   1 SUPER.SUPER      SUPER       4096 May  3  1999
/usr
```

Without the `-d` option, this LS command lists detailed information about all the files in the directory named `/usr`.

MODE Command

MODE selects the MXCI command mode.

```
MODE { MXCS | REPORT | SQL }
```

`MXCS`

specifies that commands that follow will be sent to MXCS.

`REPORT`

specifies that commands that follow will be sent to the Report Writer. For details on the Report Writer, see the *SQL/MX Report Writer Guide*.

`SQL`

specifies that commands that follow will be sent to NonStop SQL/MX. When you begin an MXCI session, the mode is set to SQL. For details on MXCI, see MXCI Session on page 1-2.

MXCI Command

MXCI is the command that starts an MXCI session from the OSS environment.

```
mxci  [-i filename] | [{ -s | -script } filename]
```

filename

specifies the file from which MXCI reads the commands.

The file must either be closed or open for read only.

The file must be an OSS text file (an odd-unstructured file, type 180) or a file specified with a Guardian path name which is an EDIT file (type 101).

If the **-i** option is specified, MXCI stops the session immediately after executing the commands in the file.

If the **-s** or **-script** option is specified, MXCI session waits for user inputs after executing the commands in the file.

-i, **-s**, and **-script** options are mutually exclusive.

Examples of MXCI Command

- Start the console version of MXCI by using the MXCI command:

  ```
  /mxutil/sys 1>mxci

  Hewlett-Packard NonStop(TM) SQL/MX Conversational Interface 2.3
  (c) Copyright 2007 Hewlett-Packard Development Company, LP.
  >>
  ```

  You can stop an MXCI session by using the EXIT command. See EXIT Command on page 4-28.

- For command:

  ```
  mxci -i input.sql
  ```

  The contents of input.sql could be:

  ```
  drop table tab;
  create table tab(a INT);
  insert into tab values(10);
  select * from tab;
  ```

The results of input file can be stored in another file using:

mxci -i *inputfile* >>*outputfile*

For example:

```
mxci -i input.sql >>result
```

- Example of MXCI command-line option **-s**

```
mxci -s inputfile.sql
```

The contents of the `inputfile.sql` can be:

```
Create catalog tempcat;
Create schema tempcat.tempsch;
Set schema tempcat.tempsch;
Create table sampletab(a int);
Insert into sampletab values(10);
Select * from sampletab;
```

The result after running the command:

```
Hewlett-Packard NonStop(TM) SQL/MX Conversational
Interface 3.2.1
(c) Copyright 2003, 2004-2013 Hewlett-Packard Development
Company, LP.
>>Create catalog tempcat;

--- SQL operation complete.
>>Create schema tempcat.tempsch;

--- SQL operation complete.
>>Set schema tempcat.tempsch;

--- SQL operation complete.
>>Create table sampletab(a int);

--- SQL operation complete.
>>Insert into sampletab values(10);

--- 1 row(s) inserted.
>>Select * from sampletab;

A

-----------

         10

--- 1 row(s) selected.

>>
```

OBEY Command

[Considerations for OBEY](#)
[Examples of OBEY](#)

The OBEY command executes MXCI commands and SQL statements from a file exactly as if you had entered the commands and statements from within an MXCI session.

You can use OBEY only within an MXCI session.

```
OBEY command-file [(section-name)]
```

`command-file`

is the path name of a file that contains MXCI commands and SQL statements to be executed by the OBEY command.

`command-file` must be an OSS text file (an odd-unstructured file, type 180) or a file specified with a Guardian path name that is an EDIT file (type 101). The file is sometimes referred to as an OBEY command file.

An OBEY command file must either be closed at the time you execute OBEY or open for read only. OBEY opens the file if necessary, executes the commands and statements, and then closes the file.

`section-name`

is the name of a section within *`command-file`* to execute.

If you specify *`section-name`*, OBEY executes the commands between the header line for the specified section and the header line for the following section (or the end of the file).

If you omit *`section-name`*, OBEY executes the entire file.

Considerations for OBEY

Specifying Sections in Command Files

Specify sections within a command file by including a section header that starts in column 1 at the beginning of each section:

```
?SECTION section-name
```

The *`section-name`* is a regular SQL identifier that is the name of the section. It cannot begin with a number or underscore. Each section name should be unique within its file, because MXCI executes only the first section it finds that has the name you specify in an OBEY command.

Effect of the MXCI Break Key

Typically, if you press the MXCI break key (Ctrl-c, Ctrl-Break, or the OutsideView Break icon) while MXCI is executing commands and statements from an OBEY command file, the current command or statement is interrupted, the processing of the OBEY command file is terminated, and the transaction might be rolled back. Execute the SHOW SESSION command to determine the status of the transaction.

Examples of OBEY

- Suppose that the EXAMPLES file is an OSS file in the current directory. Execute all statements and commands in the EXAMPLES file:

```
>>OBEY EXAMPLES;
```

- Suppose that the EXAMPLES file is a Guardian file in $DATA06.TEMPJB. Execute the statements and commands only in the SETFCNS section of the EXAMPLES file:

```
>>OBEY /G/data06/tempjb/examples (setfcns);
```

REPEAT Command

The REPEAT command re-executes a previous MXCI command.

You can use REPEAT only within an MXCI session.

```
REPEAT [text | [-]number]
```

text

specifies the text of the most recent use of a command. The command must have been executed beginning with *text*, but *text* need be only as many characters as necessary to identify the command. Leading blanks are ignored.

[-]*number*

is an integer that identifies a command in the history buffer. If *number* is negative, it indicates the position of the command in the history buffer relative to the current command; if *number* is positive, it is the ordinal number of a command in the history buffer.

The HISTORY command displays the commands or statements in the history buffer. See HISTORY Command on page 4-44.

To re-execute the immediately preceding command, enter REPEAT without specifying text or number. If you enter more than one MXCI command on a line, the REPEAT command re-executes only the last command on the line.

Examples of REPEAT

- Suppose that you have a series of statements you have executed. Re-execute the last SELECT:

```
>>REPEAT SELECT;
>>SELECT * FROM samdbcat.invent.partsupp;

PARTNUM  SUPPNUM  PARTCOST      QTY_RECEIVED
-------  -------  ------------  ------------
   2000       95       1000.00            10
   2010       99         30.00            20
   2020      186        200.00            30
   ...
```

- Re-execute the second to the last command:

```
REPEAT -2;
```

- Re-execute the second command in the history buffer:

```
REPEAT 2;
```

RESET PARAM Command

The RESET PARAM command is used to clear all parameter values or a specified parameter value within an MXCI session.

You can use RESET PARAM only within an MXCI session.

```
RESET PARAM [?param-name]
```

?param-name

is the name *param-name* of the parameter for which the value is specified. If you do not specify *param-name*, all of the parameter values in the current MXCI session are cleared. If you want to clear several parameter values but not all, you must use a separate RESET PARAM statement for each parameter.

See MXCI Parameters on page 6-77.

Examples of RESET PARAM

- Before you can execute a SELECT statement with parameters, you must specify the parameter values. Clear all parameter values so that unexpected values are not provided during execution of the FINDSUPP file.

```
RESET PARAM;

SET PARAM ?ST 'TEXAS';
SET PARAM ?PN 3210;
```

Execute the SELECT statement as follows:

```
SELECT S.suppnum, suppname
FROM sales.supplier S,
  invent.partsupp PS
WHERE S.suppnum = PS.suppnum AND
  partnum = ?PN AND state = ?ST;

SUPPNUM  SUPPNAME
-------  ------------------
     15  DATADRIVE CORP

--- 1 row(s) selected.
```

- Reset only one parameter:

Display the parameters:

```
>>SHOW PARAM;

PARAM ?ST TEXAS
PARAM ?PN 3210
```

Clear the ?ST parameter and display the parameters:

```
>>RESET PARAM ?ST;
>>SHOW PARAM;

PARAM ?PN 3210
```

Note that NonStop SQL/MX displays only the ?PN parameter.

SET LIST_COUNT Command

The SET LIST_COUNT command is used to set the maximum number of rows to be displayed in SELECT statements executed after this command.

You can use SET LIST_COUNT only within an MXCI session.

```
SET LIST_COUNT [num-rows]
```

num-rows

is a positive integer that specifies the maximum number of rows of data to be displayed from the execution of SELECT statements after the execution of this command.

To reset the number of displayed rows, issue SET LIST_COUNT without specifying the number of rows. The system-defined default setting is 4,000,000. To terminate the listing of rows, use the MXCI break key.

Considerations for SET LIST_COUNT

Range for Number of Rows

The allowable values for the list count are from 0 to the maximum value of an unsigned integer. If the specified value is 0, the number of retrieved rows is zero. If the specified value is greater than the maximum value of an unsigned integer, the number of retrieved rows is that maximum value.

Examples of SET LIST_COUNT

- Specify the number of rows to display:

```
SET LIST_COUNT 5;

SELECT empnum, first_name, last_name
FROM persnl.employee
ORDER BY empnum;

EMPNUM  FIRST_NAME       LAST_NAME
------  ---------------  --------------------
     1  ROGER            GREEN
    23  JERRY            HOWARD
    29  JANE             RAYMOND
    32  THOMAS           RUDLOFF
    39  KLAUS            SAFFERT

--- 5 row(s) selected.LIST_COUNT was reached.
```

SET PARAM Command

Considerations for SET PARAM
Examples of SET PARAM

The SET PARAM command is used to set a parameter value within an MXCI session. The value is used for queries that contain the associated parameter name. A separate SET PARAM is required for each parameter. See MXCI Parameters on page 6-77.

You can use SET PARAM only within an MXCI session.

```
SET PARAM ?param-name [[_char-set-name] param-value | NULL]
```

?*param-name*

is the name of the parameter for which the value is specified. Parameter names are case-sensitive—for example, the parameter ?pn is not equivalent to the parameter ?PN.

char-set-name

is the character set name, preceded by an underscore (_) character. Valid values are ISO88591, UCS2, KANJI or KSC5601. If you do not enter *char-set-name*, the default is ISO88591.

You can use an ISO88591 param in an SQL query as a non-character typed value (such as INT). You can use a UCS2 param in an SQL query as either a non character typed value or an ISO88591 value. You can use a param with a character set that you have specified as a character value in an SQL query where the character value is expected to be of that character set.

param-value

is a numeric or character literal that specifies the value for the parameter. If you specify *char-set-name*, you must enclose *param-value* in single quotes. Otherwise, if *param-value* is a character literal and the target column is character, you do not have to enclose it in single quotation marks; its data type is determined from the data type of the column to which the literal is assigned. If you do not specify a value, NonStop SQL/MX uses a string with a length of zero for the parameter. You can enter the value in hexadecimal format.

NULL

represents the null value. You must enter it in uppercase letters.

Considerations for SET PARAM

Using With PREPARE and EXECUTE

If you use the PREPARE statement to compile an SQL statement, you must specify all of the parameters in the prepared SQL statement with the SET PARAM command prior to issuing the EXECUTE statement.

Examples of SET PARAM

- Set param ?x to ISO88591 string 'abc':

  ```
  set param ?x _iso88591'abc'
  ```

  or

  ```
  set param ?x 'abc'
  ```

- Set param ?y to UCS2 string 'abc':

  ```
  set param ?y _ucs2'abc'
  ```

- Set param ?z to KANJI string '1234':

  ```
  set param ?z _kanji'1234'
  ```

- Set param ?x to ISO88591 string 'abc':

  ```
  set param ?x _iso88591 x'61 62 63'
  ```

- Set param ?y to UCS2 string 'abc':

  ```
  set param ?y _ucs2 x'0061 0062 0063'
  ```

- Set param ?z to KANJI string '123':

  ```
  set param ?z _kanji x'8250 8251 8253'
  ```

- Suppose that SET PARAM commands are specified as:

  ```
  SET PARAM ?ST 'TEXAS';
  SET PARAM ?PN 3210;
  ```

  Execute this query.

  ```
  SELECT S.suppnum, suppname
  FROM invent.supplier S,
    invent.partsupp PS
  WHERE S.suppnum = PS.suppnum AND
    partnum = ?PN AND state = ?ST;

  Supp/Num  Supplier Name
  --------  ------------------
        15  DATADRIVE CORP

  --- 1 row(s) selected.
  ```

  You can set values for another state and part number and rerun the query.

- The PROJECT table has a SHIP_TIMESTAMP column. This UPDATE statement uses the character literal in the ?SHIP parameter to set the value:

```
SET PARAM ?SHIP '1998-04-03 21:05:36.143';
UPDATE persnl.project
  SET ship_timestamp = CAST (?SHIP AS TIMESTAMP(3));
```

- The PROJECT table has an EST_COMPLETE column. This UPDATE statement uses the character literal in the ?EST parameter to set the value:

```
SET PARAM ?EST 60;
UPDATE persnl.project
  SET est_complete = CAST (?EST AS INTERVAL DAY);
```

SET SHOWSHAPE Command

Considerations for SET SHOWSHAPE
Examples of SET SHOWSHAPE

The SET SHOWSHAPE command allows you to display access plans in effect. The effect of SET SHOWSHAPE is to generate the output of the SHOWSHAPE command for multiple SQL statements. See SHOWSHAPE Command on page 4-110.

You can use SET SHOWSHAPE only within an MXCI session.

```
SET SHOWSHAPE showshape-option

showshape-option is:
     ON
   | OFF
   | INFILE infile-name OUTFILE outfile-name
```

`ON`

displays the access plans in effect for executed queries. The control query shape is displayed immediately before the query output.

`OFF`

turns off the display of access plans.

Note. The default setting when you start MXCI is SHOWSHAPE OFF.

`INFILE` *`infile-name`* `OUTFILE` *`outfile-name`*

allows you to specify the name of an input file of SQL statements and the name of an output file that is the result of executing the input file. The output file includes the control query shape for each query in the input file.

`infile-name`

is the full or relative path name of an input file that contains MXCI commands and SQL statements to be executed by an OBEY command. See OBEY Command on page 4-57.

`outfile-name`

is the full or relative path name of an output file to which are written the results of queries and their access plans. The query output is the result of the execution of the input OBEY command file *`infile-name`*. The control query shape for each query is displayed immediately before the query text.

Considerations for SET SHOWSHAPE

Default Control Query Shape

For those statements that do not have a shape—for example, the CREATE SCHEMA statement—a control query shape (CQS) of the form CONTROL QUERY SHAPE ANYTHING is issued.

CONTROL QUERY SHAPE ANYTHING resets the effect of any preceding CQSs. Its use is especially important when CQSs are being generated from an input file of commands and statements.

Examples of SET SHOWSHAPE

- To turn on the display of access plans, enter:

```
SET SHOWSHAPE ON;

SELECT * FROM persnl.job
WHERE jobcode >= 500;

control query shape partition_access(
scan('JOB', forward, blocks_per_access 1, mdam off));

Job/Code  Job Description
--------  ------------------
     500  ACCOUNTANT
     600  ADMINISTRATOR
     900  SECRETARY

--- 3 row(s) selected.
```

- To turn on the display of access plans, enter:

```
SET SHOWSHAPE ON;

SELECT * FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPTNUM = DEPT.DEPTNUM
  AND EMPLOYEE.LAST_NAME = 'SMITH';

control query shape merge_join(sort(
partition_access(scan('EMPLOYEE', forward,
blocks_per_access 1, mdam off))),
partition_access(scan('DEPT', forward,
blocks_per_access 3, mdam off)));

Employee/Number  First Name    Last Name      Dept/Num  ...
---------------  ------------  -------------- --------  ...
             89  PETER         SMITH              3300  ...

--- 1 row(s) selected.
```

Use this displayed plan to implement a forced plan. For more information about forcing plans, see CONTROL QUERY SHAPE Statement on page 2-62.

- To turn off the display of access plans, enter:

```
SET SHOWSHAPE OFF;
```

- To write results of queries, which are provided in an input file named `examples`, and their plans to an output file named `plans`, enter:

```
SET SHOWSHAPE INFILE /G/data06/judy/examples
              OUTFILE plans;
```

SET STATISTICS Command

The SET STATISTICS command allows you to specify whether to display statistics after each SQL statement executes.

```
SET STATISTICS {ON | OFF}
```

ON

displays statistics automatically after each statement executes.

OFF

turns off the automatic display of statistics.

Note. The default setting when you start MXCI is STATISTICS OFF.

For a description of the statistics displayed, see DISPLAY STATISTICS Command on page 4-23.

Examples of SET STATISTICS

- To enable the automatic display of statistics, enter:

```
>> SET STATISTICS ON;
>> DELETE FROM persnl.employee
+> WHERE first_name = 'TIM' AND last_name = 'WALKER';

--- 1 row(s) deleted.


Start Time               2001/08/31 09:57:33.793
End Time                 2001/08/31 09:57:37.268
Elapsed Time                        00:00:03.476
Compile Time                        00:00:02.963
Execution Time                      00:00:00.513

Table Name              Records      Records    Disk    Message       Message    Lock
                       Accessed         Used    I/Os      Count         Bytes

SAMDBCAT.PERSNL.EMPLOYEE
                             62            1       2          2         22496       0
"\MYSYS.$SAMDB".PERSNL.XEMPDEPT
                              1            1       2          2          7096       0
"\MYSYS.$SAMDB".PERSNL.XEMPNAME
                              1            1       2          2         10784       0
```

- To disable the automatic display of statistics, enter:

```
>> SET STATISTICS OFF;
>> DELETE FROM persnl.employee
+> WHERE first_name = 'GINNY' AND last_name = 'FOSTER';

--- 1 row(s) deleted.
```

SET TERMINAL_CHARSET Command

The SET TERMINAL_CHARSET command is used to set the character set for messages that an interactive SQL/MX client can send to or receive from NonStop SQL/MX. Examples of such messages include commands, SQL statements or query results generated during a MXCI session.

```
SET TERMINAL_CHARSET 'value'
```

`value`

is any one of the supported character set names (‘ISO88591’, ‘SJIS’, ‘EUCJP’, ‘BIG5’, ‘GB18030’, ‘GB2312’, ‘GBK’, ‘KSC5601’ or ‘UTF8’) enclosed in single quotes. The default value is 'ISO88591'.

You can set the character set attribute to different values during an interaction session. Its scope is limited only to the interaction session in which it is issued, and has no effect in any other contexts such as in an embedded program processed during the MXCI session. If you set the attribute to a value that does not represent a valid character set name, NonStop SQL/MX issues error 3010. If a character set name cannot be used as a terminal character set (such as the character set is not supported by any known emulators), NonStop SQL/MX issues an error.

Considerations for SET TERMINAL_CHARSET

You must verify that this attribute is compatible with the human-interface environment of the client. For example, if an OutsideView running under Japanese Windows 2000 is hosting a MXCI session, you should set the attribute to 'SJIS,' and you should use a Shift JIS compatible IME (input method editor) for input. You can receive syntax errors or garbage output if the attribute is not properly set.

SET WARNINGS Command

The SET WARNINGS command is used to turn the display of warnings on or off during an MXCI session. An MXCI session starts with the warnings on.

You can use SET WARNINGS only within an MXCI session.

```
SET WARNINGS {ON | OFF}
```

Examples of SET WARNINGS

- Suppose that T1 is a table containing one row. This example is executed with warnings on—the MXCI default:

  ```
  SET WARNINGS ON;

  SELECT CAST('abcd' as CHAR(1)) FROM T1;

  *** WARNING[8402] A string overflow occurred during the
  evaluation of a character expression.

  (EXPR)
  ------
  a

  --- 1 row(s) selected.
  ```

- Suppose that table T1 exists as in the preceding example. This example is executed with warnings off:

  ```
  SET WARNINGS OFF;

  SELECT CAST('abcd' as CHAR(1)) FROM T1;

  (EXPR)
  ------
  a

  --- 1 row(s) selected.
  ```

SH Command

SH invokes the shell of your platform.

You can use SH only within an MXCI session.

```
SH [command-line-argument]...
```

command-line-argument

To display the command-line arguments that your shell supports, at the MXCI prompt, enter:

```
sh man sh;
```

A reference page of the SH command explains the command-line arguments that your shell supports. At the end of the reference page, the MXCI prompt automatically appears.

Examples of SH

- Invoke the OSS shell from MXCI:

```
>>SH;
/usr/jbrook:
```

- Return to MXCI from the OSS shell:

```
/usr/jbrook:exit

>>
```

SHOW PARAM Command

The SHOW PARAM command is used to display all of the parameters and their values that are defined in the current MXCI session.

You can use SHOW PARAM only within an MXCI session.

```
SHOW PARAM
```

See MXCI Parameters on page 6-77.

Examples of SHOW PARAM

- Display parameter values:

  ```
  SHOW PARAM;

  Param ?ST TEXAS
  Param ?PN 3210
  Param ?pn 1234
  ```

  Note that parameter names are case-sensitive. For example, the parameter ?pn is not equivalent to the parameter ?PN. The two parameters have different values.

SHOW PREPARED Command

The SHOW PREPARED command is used to display prepared statements in the current MXCI session.

Use SHOW PREPARED only within an MXCI session.

```
SHOW PREPARED [* | ALL]
```

`* | ALL`

displays all the currently prepared statements.

Note. The SHOW PREPARED command displays all the currently prepared statements regardless of whether you specify the `*` or the `ALL` option.

See An operation is a postfix merge if the range of data ends at the bottom of the partition. You can specify only the TO NEXT PARTITION clause. The split partition cannot be the last partition (the rightmost partition in the list). on page 2-279.

Examples of SHOW PREPARED

- Display all currently prepared statements:

```
>>SHOW PREPARED;

FINDEMP
  SELECT * FROM PERSNL.EMPLOYEE WHERE SALARY > 40000.00
  AND JOBCODE = 450;

EMPCOM
  SELECT FIRST_NAME, LAST_NAME, DEPTNUM FROM PERSNL.EMPLOYEE
  WHERE DEPTNUM <> 1500 AND SALARY <= (SELECT AVG(SALARY)
  FROM PERSNL.EMPLOYEE WHERE DEPTNUM = 1500);
```

- This command also displays all currently prepared statements:

```
>>SHOW PREPARED *;

FINDEMP
  SELECT * FROM PERSNL.EMPLOYEE WHERE SALARY > 40000.00
  AND JOBCODE = 450;

EMPCOM
  SELECT FIRST_NAME, LAST_NAME, DEPTNUM FROM PERSNL.EMPLOYEE
  WHERE DEPTNUM <> 1500 AND SALARY <= (SELECT AVG(SALARY)
  FROM PERSNL.EMPLOYEE WHERE DEPTNUM = 1500);
```

SHOW SESSION Command

Examples of SHOW SESSION

SHOW SESSION displays attributes of the current MXCI session. You can use SHOW SESSION (or ENV) only within an MXCI session.

```
SHOW SESSION
```

SHOW SESSION displays these attributes:

CURRENT DIRECTORY	Path name of the current server directory. You can use the CD command to change it.
HOME DIRECTORY	Default directory.
LIST_COUNT	Current list count.
LOG FILE	Current log file.
MESSAGEFILE	Current message file.
TERMINAL_CHARSET	Current character set for the session.
MESSAGEFILE LANG	Language of the text in the message file.
MESSAGEFILE VRSN	Version of the message file; its value is stored in the message.
SQL CATALOG	Default catalog.
SQL SCHEMA	Default schema.
TRANSACTION ID	Transaction identifier of the current transaction if one is in progress.
TRANSACTION STATE	Transaction status (in progress or not in progress).
WARNINGS	Current state of warnings (on or off).

Examples of SHOW SESSION

- A SHOW SESSION command and its output:

```
>>SHOW SESSION
+>;
----------------------------------
Current Environment
----------------------------------
CURRENT DIRECTORY  /usr/manager/bin
HOME DIRECTORY     /usr/manager
LIST_COUNT         5
LOG FILE
MESSAGEFILE        /usr/manager/bin/mxcierrors.cat
TERMINAL CHARSET   ISO88591
MESSAGEFILE LANG   US English
MESSAGEFILE VRSN   {2003-12-11 13:56 NSK:SQUAW/SUPER.SUPER}
SQL CATALOG        CAT
SQL SCHEMA         SCH
TRANSACTION ID
TRANSACTION STATE  not in progress
WARNINGS           on
```

SHOWCONTROL Command

Examples of SHOWCONTROL

The SHOWCONTROL command displays the access plan, controls, and system defaults in effect.

Use SHOWCONTROL only within an MXCI session.

```
SHOWCONTROL showcontrol-option

showcontrol-option is:
     [QUERY] SHAPE
   | TABLE [table [,MATCH {FULL | PARTIAL}]]
   | [QUERY] DEFAULT [attribute-name [,MATCH {FULL |
                                             PARTIAL}]]
   | ALL
```

`[QUERY] SHAPE`

displays the access plan (or control query shape) in effect, which is the result of the last CONTROL QUERY SHAPE statement that is executed. See CONTROL QUERY SHAPE Statement on page 2-62.

`TABLE`

displays all controls in effect that are the result of CONTROL TABLE statements. See CONTROL TABLE Statement on page 2-74.

`table` `[,MATCH {FULL | PARTIAL}]`

displays only the table controls in effect that match, either fully or partially, the *`table`* used in CONTROL TABLE statements. The match is not case-sensitive.

MATCH FULL specifies that *`table`* must be the same as the table name used in CONTROL TABLE statements. MATCH PARTIAL specifies that *`table`* must be included in the table name used in CONTROL TABLE statements. The default is MATCH PARTIAL.

`[QUERY] DEFAULT`

displays all system defaults in effect that are the result of executing CONTROL QUERY DEFAULT statements or executing other statements that also affect the external system defaults—for example, SET CATALOG. See CONTROL QUERY DEFAULT Statement on page 2-60.

`attribute-name` `[,MATCH {FULL | PARTIAL}]`

displays only the system defaults in effect that match, either fully or partially, the *`attribute`* used in CONTROL QUERY DEFAULT statements. The match is not case-sensitive.

MATCH FULL specifies that *attribute-name* must be the same as the attribute name used in a CONTROL QUERY DEFAULT statement. MATCH PARTIAL specifies that *attribute-name* must be included in the attribute name used in a CONTROL QUERY DEFAULT statement. The default is MATCH PARTIAL.

If *attribute-name* is a reserved word, such as MAX, MIN, or TIME, you must capitalize *attribute-name* and delimit it within double quotes ("). The only exceptions to this rule are the reserved words CATALOG and SCHEMA, which you can either capitalize and delimit within double quotes or specify without quotation marks.

`ALL`

displays all CONTROL QUERY SHAPE settings, CONTROL TABLE settings, CONTROL QUERY DEFAULT settings, and a list of all default settings in effect. See System Defaults Table on page 10-37.

Examples of SHOWCONTROL

- Show the access plan in effect when CONTROL QUERY SHAPE has not been executed in the current session:

```
SHOWCONTROL SHAPE;

No CONTROL QUERY SHAPE settings are in effect.

--- SQL operation complete.
```

- Issue one or more CONTROL QUERY SHAPE statements followed by a SHOWCONTROL SHAPE:

```
CONTROL QUERY SHAPE NESTED_JOIN(PARTITION_ACCESS(SCAN('J',
FORWARD, MDAM OFF)),MATERIALIZE(PARTITION_ACCESS(SCAN('E',
FORWARD, MDAM OFF))));

--- SQL operation complete.

CONTROL QUERY SHAPE  NESTED_JOIN (PARTITION_ACCESS(SCAN),
PARTITION_ACCESS(SCAN('DEPT')));

--- SQL operation complete.

SHOWCONTROL SHAPE;

CONTROL QUERY SHAPE  NESTED_JOIN (PARTITION_ACCESS(SCAN),
  PARTITION_ACCESS(SCAN('DEPT')));

--- SQL operation complete.
```

- Display output when CONTROL TABLE has not been executed in the current session:

```
SHOWCONTROL TABLE;

No CONTROL TABLE settings are in effect.

--- SQL operation complete.
```

- Issue multiple CONTROL TABLE statements followed by a SHOWCONTROL TABLE:

```
CONTROL TABLE PERSNL.JOB MDAM 'OFF';

--- SQL operation complete.

CONTROL TABLE * TIMEOUT '3000';

--- SQL operation complete.

SHOWCONTROL TABLE;
CONTROL TABLE SAMDBCAT.PERSNL.JOB
  MDAM                        OFF
CONTROL TABLE *
  TIMEOUT                     3000

--- SQL operation complete.
```

- Issue multiple CONTROL QUERY DEFAULT statements followed by a SHOWCONTROL DEFAULT:

```
SET CATALOG SAMDBCAT;

CONTROL QUERY DEFAULT ISOLATION_LEVEL 'READ UNCOMMITTED';

--- SQL operation complete.

CONTROL QUERY DEFAULT TIMEOUT '1000';

--- SQL operation complete.

SHOWCONTROL DEFAULT;

CONTROL QUERY DEFAULT
  CATALOG                     SAMDBCAT
  ISOLATION_LEVEL             READ UNCOMMITTED
  TIMEOUT                     1000

--- SQL operation complete.
```

Note that the catalog name is set by the SET CATALOG statement.

- Change the TIMEOUT attribute and then issue a SHOWCONTROL DEFAULT for TIME, which is a reserved word:

```
CONTROL QUERY DEFAULT TIMEOUT '2000';

--- SQL operation complete.

SHOWCONTROL DEFAULT "TIME", MATCH PARTIAL;

CONTROL QUERY DEFAULT
  TIMEOUT                      2000

Current DEFAULTS
  STREAM_TIMEOUT               -1
  TIMEOUT                      2000

--- SQL operation complete.
```

  In this example, the TIME name matches the TIMEOUT and STREAM_TIMEOUT attributes.

- Change the CATALOG attribute and then issue a SHOWCONTROL DEFAULT for CAT:

```
CONTROL QUERY DEFAULT CATALOG 'SAMDBCAT';

--- SQL operation complete.

SHOWCONTROL DEFAULT CAT;

CONTROL QUERY DEFAULT
  CATALOG                        SAMDBCAT

Current DEFAULTS
  CATALOG                        SAMDBCAT
  SCHEMA                         PERSNL

  --- SQL operation complete.
```

  In this example, the CAT name matches only the CATALOG attribute. Note that the SCHEMA attribute is always displayed if the CATALOG attribute is displayed, and the reverse is also true.

- Display all settings and defaults in effect:

```
>>showcontrol all;
..

No CONTROL QUERY SHAPE settings are in effect.

No CONTROL TABLE settings are in effect.

No CONTROL QUERY DEFAULT settings are in effect.

Current DEFAULTS
  ATTEMPT_ASYNCHRONOUS_ACCESS          ON
  ATTEMPT_ESP_PARALLELISM              SYSTEM
  AUTOMATIC_RECOMPILATION              ON
  CACHE_HISTOGRAMS                     ON
  CACHE_HISTOGRAMS_REFRESH_INTERVAL    3600
  CATALOG                              CAT
  CHECK_CONSTRAINT_PRUNING             ON
  CREATE_DEFINITION_SCHEMA_VERSION     SYSTEM
  CROSS_PRODUCT_CONTROL                ON
  DATA_FLOW_OPTIMIZATION               ON
  DDL_DEFAULT_LOCATIONS
  DEF_MAX_HISTORY_ROWS                 1024
  DOOM_USERTRANSACTION                 OFF
  DP2_CACHE_4096_BLOCKS                1024
  DYNAMIC_HISTOGRAM_COMPRESSION        ON
  FFDC_DIALOUTS_FOR_MXCMP              OFF
  FLOATTYPE                            IEEE
  GENERATE_EXPLAIN                     ON
  GEN_EIDR_BUFFER_SIZE                 31000
  GEN_MAX_NUM_PART_DISK_ENTRIES        3
  GEN_MAX_NUM_PART_NODE_ENTRIES        255
  GEN_PA_BUFFER_SIZE                   31000
  HIST_BASE_REDUCTION                  ON
  HIST_DEFAULT_SEL_FOR_LIKE_WILDCARD   0.10
  HIST_DEFAULT_SEL_FOR_PRED_RANGE      0.3333
  HIST_JOIN_CARD_LOWBOUND              1.0
  HIST_NO_STATS_REFRESH_INTERVAL       3600
  HIST_NO_STATS_ROWCOUNT               100
  HIST_NO_STATS_UEC                    2
  HIST_PREFETCH                        ON
  HIST_ROWCOUNT_REQUIRING_STATS        50000
  HIST_SAME_TABLE_PRED_REDUCTION       0.0
  HIST_SCRATCH_VOL
  HIST_SECURITY_WARNINGS               ON
  INDEX_ELIMINATION_LEVEL              MAXIMUM
  INFER_CHARSET                        OFF
  INSERT_VSBB                          SYSTEM
  INTERACTIVE_ACCESS                   OFF
  ISOLATION_LEVEL                      READ_COMMITTED
  IUD_NONAUDITED_INDEX_MAINT           OFF
  JOIN_ORDER_BY_USER                   OFF
  MATERIALIZE                          SYSTEM
  MAX_ESPS_PER_CPU_PER_OP              1
```

```
MAX_ROWS_LOCKED_FOR_STABLE_ACCESS     1
MDAM_SCAN_METHOD                      ON
MIN_MAX_OPTIMIZATION                  ON
MIN_COALITIONS
MP_SUBVOLUME                          SKYOSTST
MP_SYSTEM                             \HPIDMR5
MP_VOLUME                             $DATA01
MSCF_ET_REMOTE_MSG_TRANSFER           0.00005
MULTIUNION                            ON
MV_AS_ROW_TRIGGER                     OFF
MV_REFRESH_MAX_PARALLELISM            1
MV_REFRESH_MAX_PIPELINING             1
MXCMP_PLACES_LOCAL_MODULES            OFF
NAMETYPE                              ANSI
NATIONAL_CHARSET                      UCS2
NOT_NULL_CONSTRAINT_DROPPABLE_OPTION  OFF
NUMBER_OF_USERS                       1
OLT_QUERY_OPT                         ON
OPTIMIZATION_LEVEL                    3
OPTS_PUSH_DOWN_DAM                    0
PARALLEL_NUM_ESPS                     SYSTEM
PM_OFFLINE_TRANSACTION_GRANULARITY    5000
PM_ONLINE_TRANSACTION_GRANULARITY     400
POS_LOCATIONS
POS_NUM_OF_PARTNS                     0
POS_RAISE_ERROR                       0
PREFERRED_PROBING_ORDER_FOR_NESTED_JOIN        OFF
PRESERVE_MIN_SCALE                    0
PRIMARY_KEY_CONSTRAINT_DROPPABLE_OPTION        OFF
QUERY_CACHE                           1024
QUERY_CACHE_MAX_VICTIMS               10
QUERY_CACHE_REQUIRED_PREFIX_KEYS      255
QUERY_CACHE_STATEMENT_PINNING         OFF
READONLY_CURSOR                       TRUE
RECOMPILATION_WARNINGS                OFF
REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT         SYSTEM
REMOTE_ESP_ALLOCATION                 SYSTEM
SAVE_DROPPED_TABLE_DDL                OFF
SCHEMA                                SCH
SCRATCH_DISKS
SCRATCH_DISKS_EXCLUDED
SCRATCH_DISKS_PREFERRED
SCRATCH_FREESPACE_THRESHOLD_PERCENT   10
SIMILARITY_CHECK                      ON
SORT_MAX_HEAP_SIZE_MB                 20
STREAM_TIMEOUT                        -1
TABLELOCK                             SYSTEM
TEMPORARY_TABLE_HASH_PARTITIONS
TIMEOUT                               6000
UDR_JAVA_OPTIONS                      OFF
UNION_TRANSITIVE_PREDICATES           ON
UPD_ORDERED                           ON
UPD_SAVEPOINT_ON_ERROR                ON
VARCHAR_PARAM_DEFAULT_SIZE            255
ZIG_ZAG_TREES                         OFF
```

```
--- SQL operation complete.
>>
```

SHOWDDL Command

[Considerations for SHOWDDL](#)
[Examples of SHOWDDL](#)

The SHOWDDL command displays the DDL syntax used to create a table, view, or stored procedure as it exists in metadata and optionally lists the object's dependent objects. You can use the SHOWDDL output as input to MXCI to recreate the specified object.

SHOWDDL is an SQL/MX extension.

```
SHOWDDL {[PROCEDURE] procedure-name }| {object-name
[,SQLMP]|[,DEPENDENT objects]]}

procedure-name is:

  [[catalog-name.]schema-name.]procedure-name

object-name is:

  [[catalog-name.]schema-name.]object-name.

objects is:

 { TABLES | CONSTRAINTS }
```

procedure-name

specifies the name of a stored procedure. If you do not fully qualify *procedure-name*, SHOWDDL uses the default catalog and schema for the session.

object-name

specifies the ANSI name of a table, view, or SQL/MP alias. If you do not fully qualify *object-name*, SHOWDDL uses the default catalog and schema for the session.

SQLMP

specifies that SQL/MP DDL is to be generated. The default is to generate DDL for an SQL/MP object in SQL/MX syntax.

Note. The SQLMP option is applicable only for SQL/MP tables.

`DEPENDENT TABLES`

displays the list of tables with referential integrity constraints that reference this table.

`DEPENDENT CONSTRAINTS`

displays the list of referential integrity constraints for this table.

The list of dependent objects is displayed as commented lines in the SHOWDDL output.

Considerations for SHOWDDL

- SHOWDDL cannot accurately replicate the original creation text for an object.

  For ways in which the output of SHOWDDL can differ from the original DDL used to create an object, see Differences Between SHOWDDL Output and Original DDL on page 4-85.

- SHOWDDL will start a TMF transaction if one is not present. Ensure that TMF is running for SHOWDDL to start the transaction.
- SHOWDDL output is in the English (ISO88591) character set.
- When used on an SQL/MP table through an SQL/MP alias, SHOWDDL displays the DDL of the SQL/MP table using equivalent SQL/MX syntax.
- SHOWDDL can display referential integrity actions. If the referential integrity action is NO ACTION, then it is not displayed in the output.
- The SHOWDDL DEPENDENT option is not supported for views and triggers.
- The SHOWDDL output shows if the view Similarity Check option is enabled or disabled for views.
- Starting with SQL/MX Release 3.2.1, SHOWDDL displays the DDL syntax for sequence generators. SHOWDDL also displays the Guardian file location of SG_TABLE associated with the sequence generator.
- SHOWDDL will display the following attributes for the IDENTITY column:
  - Default specification of the IDENTITY column
  - Internal Sequence Generator attributes
  - Location of the SG Table

Differences Between SHOWDDL Output and Original DDL

- SHOWDDL displays SQL/MX system-created indexes as user-created indexes. In the output of SHOWDDL, each system-created index is preceded by the comment `'--The following index is a system-created index--'`. Because you cannot explicitly create a system-created index, feeding the output of a system-created index back into MXCI results in a user-created index.
- All column constraints (NOT NULL, UNIQUE, PRIMARY KEY, CHECK, REFERENCES) are transformed into table constraints. For NOT NULL constraints, “NOT NULL [NOT DROPPABLE]” is included in the column definitions but is commented out. All NOT NULL NOT DROPPABLE constraints are consolidated into a single check constraint, while NOT NULL DROPPABLE column constraints remain in separate check constraints.
- Each droppable constraint that creates an index (droppable primary key and unique constraints) is moved out of the CREATE TABLE statement and

encapsulated in a separate ALTER TABLE ADD CONSTRAINT statement. Creating an index before creating the constraint that is dependent on the index allows the details of the index to be specified explicitly.

- Check constraints are moved out of the CREATE TABLE statement and encapsulated in a separate ALTER TABLE ADD CONSTRAINT statement.
- In cases where an index is created by the system to support a not droppable primary key constraint, the DDL of this system-created index is commented out (each line is preceded by "--"). Unlike droppable constraints, a not droppable primary key constraint affects the structure of a table and therefore cannot be moved from the CREATE TABLE statement and into an ALTER TABLE ADD CONSTRAINT statement.

  Consequently, if a system-created index is implicitly created by the system to support a not droppable primary key constraint, the DDL output for explicitly creating such an index must be commented out, or a duplicate index results.
- SHOWDDL generates ALTER TABLE ADD COLUMN statements for each column that was added to the table. SHOWDDL also generates the comment `'--The partition is offline-'` before the DDL of each partition that is offline because of a partition management operation, and the DDL for the offline partitions is commented out. The entire partition clause is commented out if all of the partitions are offline.
- The PIC data type is stored as CHAR, DECIMAL, or NUMERIC in NonStop SQL/MX. SHOWDDL, therefore, displays these data types in place of the PIC data type.
- The NCHAR data type is displayed as a CHAR CHARACTER SET *`default-char-set`* showing the current default national character set (either UCS2 or ISO88591.)
- All ANSI names in the output are fully qualified.
- All physical location names are fully expanded.
- SHOWDDL displays constraint names even though they might not have been specified during the creation of the constraint.
- STORE BY is displayed even though it might not have been explicitly stated in the creation of the table.
- The ordering of the primary key (ASC/DESC) might differ from that of the original DDL because it might be changed by the STORE BY ordering.
- If NO HEADING is specified for a column, NonStop SQL/MX stores it as HEADING `''` (blank,) which SHOWDDL displays.
- If the column name, which is stored as an upshifted string unless it is delimited, is identical to the heading (case-sensitive), NonStop SQL/MX treats it as if no heading was entered. SHOWDDL does not display a heading.

- If there are two not null droppable constraints on the same user added column, only one of these is displayed.
- The ALLOCATE attribute is not stored in metadata or file label, so it is not displayed.
- The partitioning key is displayed only if it is different from the store by key. Such a scenario is when SYSKEY is a part of the store by key but not a part of the partitioning key, although the keys might appear to be the same because the SYSKEY is not displayed by SHOWDDL.
- SHOWDDL does not omit the optional clauses of the CREATE PROCEDURE statement, such as LOCATION, CONTAINS SQL, NOT DETERMINISTIC, and NO ISOLATE.
- SHOWDDL always generates a Java signature for the SPJ.
- SHOWDDL does not display the GRANT and REVOKE statements used to grant or revoke any privileges on the table.

SQL/MP Conversion Issues

Note these syntax conversions when you are displaying DDL for an SQL/MP table:

- If you run SHOWDDL on an MP alias, the MP alias name is displayed as the table name unless you use the SQLMP option, in which case the SQL/MP table name is displayed. SHOWDDL fully qualifies all SQL/MP aliases and fully expands SQL/MP table names.
- The subvolume name and table name in the physical location of the SQL/MP table are invalid for the SQL/MX syntax. If you do not specify the SQLMP option, only system name and volume name of the SQL/MP physical location are displayed with the location clause for SQL/MP tables.
- The SMF logical name is displayed instead of the physical volume name (PHYSVOL) for SQL/MP tables located on SMF volumes.
- SHOWDDL does not display individual ALTER TABLE ADD COLUMN statements for added columns in SQL/MP tables as they are for SQL/MX tables. The message [-- This SQL/MP table contains user added columns --] is displayed before the DDL of the table, and all columns are included in the DDL for the table.
- NATIONAL CHAR (NCHAR) data type is converted into CHAR CHARACTER SET using the default national character set.
- Character sets that NonStop SQL/MP supports but NonStop SQL/MX does not support are displayed by SHOWDDL, but the warning "*** WARNING[3010] Character set ISO88599 is not yet supported." is displayed.
- The UNKNOWN character set in NonStop SQL/MP is converted into ISO88591 if you do not specify the SQLMP option.

- Only COLLATE DEFAULT is supported by NonStop SQL/MX. Other collations that are supported by NonStop SQL/MP are displayed but are not valid for NonStop SQL/MX.
- The COLLATE statement must come last in an SQL/MP column definition or an SQL/MP syntax error occurs, even though this is valid SQL/MP syntax. However, this is not the order in which SHOWDDL outputs, so if you use SHOWDDL output as input for NonStop SQL/MP you will receive this syntax error:

  ```
  DEFAULT NULL is displayed after COLLATE.
  ```
- FLOAT data type can be converted into equivalent REAL data types with a precision value.
- UPSHIFT is not displayed for PIC X data type.
- For NUMERIC and SMALLINT data types, SIGNED does not appear because it is the default. Only UNSIGNED is displayed.
- DATETIME is not a supported data type in NonStop SQL/MX, but SHOWDDL displays this data type for SQL/MP tables that contain it.
- The largest MAXEXTENTS value for an SQL/MX table is 768, but it is 959 for NonStop SQL/MP.
- SHOWDDL displays the EXTENT and MAXEXTENTS only for the primary partition of an SQL/MP table or index.
- The only allowed BLOCKSIZE supported by NonStop SQL/MX is 4096. If you do not specify the SQLMP option and an SQL/MP table has a BLOCKSIZE other than 4096, its BLOCKSIZE is still displayed as 4096 for SQL/MX syntax. If its BLOCKSIZE is 4096, it is not displayed because this is the default.
- SHOWDDL displays only whether an SQL/MP table has DCOMPRESS on or off and does not distinguish between compression methods 1 and 2. DCOMPRESS is displayed only with the SQLMP option.
- KEYTAG is displayed as an unsigned small integer because of how it is stored. KEYTAG is entered as two bytes of CHAR data, but SHOWDDL shows the converted values. KEYTAG is displayed only with the SQLMP option.
- RECLENGTH is not supported because it applies only to relative sequenced files which are not supported by NonStop SQL/MX.
- DSLACK, ISLACK, and SLACK for indexes are not displayed by SHOWDDL.
- If you do not specify the SQL/MP option, SQL/MP NOT NULL column constraints are converted to NOT NULL NOT DROPPABLE constraints.
- SHOWDDL on an SQL/MP view includes added correlated names:

  ```
  CREATE VIEW V1 ( N ) AS SELECT N FROM
  \FIGARO.$DATA05.DEANCAT.T1 T1 ;
  ```

When displaying an SQL/MP view in SQL/MX syntax (not using SQLMP option), you must manually remove correlated Guardian location names because they are not valid SQL/MX syntax:

```
CREATE VIEW V1 ( N ) AS SELECT N FROM T1 T1 ;
```

- Headings for SQL/MP views are not supported.
- Added check not null constraints do not have accompanying "-- NOT NULL " comments by columns that they determine are not null, as in the output for an SQL/MX table. This situation is caused by differences in how NonStop SQL/MP and NonStop SQL/MX implement NOT NULL constraints.
- SHOWDDL does not display any table or column privilege information for the table.
- If you do not specify the SQL/MP option, SHOWDDL displays the string constants enclosed with single quotes.

Examples of SHOWDDL

- This is an example of SHOWDDL on an SQL/MX table that contains unique and primary key constraints:

```
>>CREATE TABLE CAT.SCH.T1
(N INT NOT NULL,
C INT NOT NULL UNIQUE,
CONSTRAINT PK PRIMARY KEY (N) NOT DROPPABLE)
STORE BY (C DESC, N)
ATTRIBUTE MAXEXTENTS 600;

>>SHOWDDL CAT.SCH.T1;

CREATE TABLE CAT.SCH.T1
 (
   N       INT NO DEFAULT -- NOT NULL NOT DROPPABLE
 , C       INT NO DEFAULT -- NOT NULL NOT DROPPABLE
 , CONSTRAINT CAT.SCH.PK PRIMARY KEY (N ASC) NOT DROPPABLE
 , CONSTRAINT CAT.SCH.T1_102261179_0003 CHECK
      (CAT.SCH.T1.N IS NOT NULL AND
    CAT.SCH.T1.C IS NOT NULL) NOT DROPPABLE
 )
 LOCATION \FIGARO.$DATA1.ZSDQXXBK.B7VVVW00
 NAME FIGARO_DATA1_ZSDQXXBK_B7VVVW00
  ATTRIBUTES MAXEXTENTS 600
 STORE BY (C DESC, N ASC)
 ;
-- The following index is a system created index --
CREATE UNIQUE INDEX T1_102261179_0004 ON CAT.SCH.T1
  (
    C ASC
  )
  LOCATION \FIGARO.$DATA2.ZSDUXXBK.B7VVVW00
  NAME FIGARO_DATA2_ZSDQXXBK_B7VVVW00
  ATTRIBUTES MAXEXTENTS 600
  ;
-- The following index is a system created index --
--CREATE UNIQUE INDEX PK ON CAT.SCH.T1
--  (
--    N ASC
--  )
--  LOCATION \FIGARO.$DATA1.ZSDXXXBK.B7VVVW00
--  NAME FIGARO_DATA1_ZSDQXXBK_B7VVVW00
--  ATTRIBUTES MAXEXTENTS 600
--  ;
ALTER TABLE CAT.SCH.T1
 ADD CONSTRAINT CAT.SCH.T1_102261179_0004 UNIQUE
(C) DROPPABLE;
```

Note how the unique constraint is moved out of the CREATE TABLE statement and into an ALTER TABLE statement, how the index supporting the unique constraint precedes the creation of the unique constraint, and how the index supporting the not droppable primary key is commented out because a system created index would be implicitly created.

- These are examples of SHOWDDL on tables with partitions that are offline. Note the commenting out of partitions that are offline, and the whole partition clause if all of the partitions are offline.

```
>>SHOWDDL T1;

CREATE TABLE CAT.SCH.T1
  (
    A    INT DEFAULT NULL
  , B   INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , C  INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , CONSTRAINT CAT.SCH.T1_104871912_0091
PRIMARY KEY (B ASC, C DESC) NOT DROPPABLE
  , CONSTRAINT CAT.SCH.T1_104871912_0090 CHECK
(CAT.SCH.T1.B IS NOT NULL AND
      CAT.SCH.T1.C IS NOT NULL) NOT DROPPABLE
  )
  LOCATION \FIGARO.$DATA.ZSDADM53.VZBRLI00
  NAME FIGARO_DATA_ZSDADM53_VZBRLI00
--  HASH PARTITION
--  (
-- The following partition is offline --
--    ADD LOCATION \FIGARO.$DATA.ZSDWWWWW.AADZ1200
--    NAME FIGARO_$DATA_ZSDWWWWW_AADZ1200
--  )
  STORE BY (B ASC, C DESC)
  ;

>>showddl t1;

CREATE TABLE CAT.SCH.T1
  (
    C1     INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , C2    INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , C3    INT DEFAULT NULL
  , CONSTRAINT CAT.SCH.T1_104871911_0089
      PRIMARY KEY (C1 ASC, C2 ASC) NOT DROPPABLE
  , CONSTRAINT CAT.SCH.T1_104871911_0088 CHECK
     (CAT.SCH.T1.C1 IS NOT NULL AND
      CAT.SCH.T1.C2 IS NOT NULL) NOT DROPPABLE
  )
  LOCATION \FIGARO.$DATA.ZSDADM53.QUSGEI00
  NAME FIGARO_DATA_ZSDADM53_QUSGEI00
  PARTITION
  (
    ADD FIRST KEY (1200)
      LOCATION \FIGARO.$DATA.ZSDWWWWW.AADZ1400
-- The following partition is offline --
--    ADD FIRST KEY (1300)
--      LOCATION \FIGARO.$DATA.ZSDWWWWW.AADZ1600
--      NAME FIGARO_$DATA_ZSDWWWWW_AADZ1600
  , ADD FIRST KEY (1500)
      LOCATION \FIGARO.$DATA.ZSDWWWWW.AADZ1200
      NAME FIGARO_$DATA_ZSDWWWWW_AADZ1200
  )
```

```
  STORE BY (C1 ASC, C2 ASC)
  ;
```

- This is an example of SHOWDDL on an SQL/MP table. By default, SQL/MX syntax is used to output the DDL of the table, and file names are not fully qualified.

```
>>CREATE TABLE T1 (NAME CHAR(10) DEFAULT "NOBODY"
HEADING "NAME",
                 SID LARGEINT NOT NULL,
                 PRIMARY KEY (SID DESC),
                 SSN INT UNSIGNED NOT NULL,
                 BIRTHDATE DATE NOT NULL
                 )
                 EXTENT 48
                 MAXEXTENTS 300
                 PARTITION (
                           \FIGARO.$DATA14.DEANCAT.T1
                           FIRST KEY (5000),
                           \FIGARO.$DATA15.DEANCAT.T1
                           EXTENT 48
                           MAXEXTENTS 300
                           FIRST KEY (10000))
                  NO AUDITCOMPRESS;

--- SQL operation complete.

>>CREATE INDEX IDXA on T1 (SID)
PARTITION (\FIGARO.$DATA14.DEANCAT.IDXA FIRST KEY(100),
\FIGARO.$DATA15.DEANCAT.IDXA FIRST KEY (1000));

--- SQL operation complete.

>>CREATE UNIQUE INDEX UIDX ON T1 (SSN);

--- SQL operation complete.

>>CREATE CONSTRAINT C1 on T1 CHECK (NAME >"AAA");

--- SQL operation complete.



>>SHOWDDL T1;

CREATE TABLE T1
  (
    NAME   CHAR(10) CHARACTER SET ISO88591 COLLATE
      DEFAULT DEFAULT 'NOBODY' HEADING 'NAME'
  , SID  LARGEINT NO DEFAULT NOT NULL NOT DROPPABLE
  , SSN  INT UNSIGNED NO DEFAULT NOT NULL NOT DROPPABLE
  , BIRTHDATE DATE NO DEFAULT NOT NULL NOT DROPPABLE
  , PRIMARY KEY (SID DESC)
  )
  LOCATION \FIGARO.$DATA17
  ATTRIBUTES NO AUDITCOMPRESS
```

```
    , EXTENT (48, 48),MAXEXTENTS 300
    PARTITION
    (
      ADD FIRST KEY (10000)
        LOCATION \FIGARO.$DATA14
    , ADD FIRST KEY (5000)
        LOCATION \FIGARO.$DATA15
    )
    ;
  CREATE INDEX IDXA ON T1
    (
      SID ASC
    )
    LOCATION \FIGARO.$DATA17
    PARTITION
   (
     ADD FIRST KEY (100)
        LOCATION \FIGARO.$DATA14
   , ADD FIRST KEY (1000)
        LOCATION \FIGARO.$DATA15
    )
    ATTRIBUTES NO AUDITCOMPRESS

    ;
  CREATE UNIQUE INDEX UIDX ON T1
    (
      SSN ASC
    )
    LOCATION \FIGARO.$DATA17
   ATTRIBUTES NO AUDITCOMPRESS
    ;
  ALTER TABLE T1
    ADD CONSTRAINT C1 CHECK (NAME >'AAA') ;

  --- SQL operation complete.
```

- This is an example of SHOWDDL on an SQL/MP table using the SQLMP syntax option. File names are shown fully qualified.

```
>>CREATE TABLE T1 (name CHAR(10) DEFAULT 'nobody'
HEADING 'NAME',
                 SID LARGEINT NOT NULL,
                 PRIMARY KEY (SID DESC),
                 SSN INT UNSIGNED NOT NULL,
                 birthdate DATE NOT NULL
                 )
                 PARTITION (
                           \FIGARO.$DATA14.DEANCAT.T1
                           EXTENT 32
                           MAXEXTENTS 300
                           FIRST KEY 5000)
                   NO AUDITCOMPRESS;

>>CREATE CONSTRAINT C1 on T1 CHECK SID > 1000;
```

```
>>SHOWDDL T1, SQLMP
CREATE TABLE \FIGARO.$DATA17.DEANCAT.T1
  (
    NAME  CHAR(10) CHARACTER SET ISO88591 COLLATE
      DEFAULT DEFAULT 'nobody' HEADING 'NAME'
  , SID  LARGEINT NO DEFAULT
      -- NOT NULL NOT DROPPABLE
  , SSN INT UNSIGNED NO DEFAULT
      -- NOT NULL NOT DROPPABLE
  , BIRTHDATE DATE NO DEFAULT -- NOT NULL NOT DROPPABLE
  , PRIMARY KEY (SID DESC)
  )

  CATALOG \FIGARO.$DATA17.DEANCAT
  PARTITION (
    \FIGARO.$DATA14.DEANCAT.T1
    FIRST KEY 5000
  )
  ;
CREATE CONSTRAINT C1 on \FIGARO.$DATA17.DEANCAT.T1 CHECK SID
> 1000;
```

- This is an example of SHOWDDL on a table with a trigger. The DDL of the triggers is shown, but the ALTER TRIGGER DISABLE statement is not displayed for triggers that are disabled.

```
>>CREATE TABLE T074T3
(A INT NOT NULL, B INT, C CHAR(8), D INT, PRIMARY KEY(A));

>>CREATE TRIGGER BTR BEFORE UPDATE ON T074T3
  REFERENCING OLD AS MYOLDROW,
  NEW AS MYNEWROW WHEN (MYNEWROW.D > MYOLDROW.D)
  SET MYNEWROW.B = MYNEWROW.B + MYOLDROW.D;

>>SHOWDDL T074T3;

CREATE TABLE CAT.SCH.T074T3
  (
    A        INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , B        INT DEFAULT NULL
  , C       CHAR(8) CHARACTER SET ISO88591 COLLATE
      DEFAULT DEFAULT NULL
  , D            INT DEFAULT NULL
  , CONSTRAINT CAT.SCH.T074T3_102459148_0001
PRIMARY KEY (A ASC) NOT DROPPABLE
  , CONSTRAINT CAT.SCH.T074T3_102459148_0000
CHECK (CAT.SCH.T074T3.A IS NOT
      NULL) NOT DROPPABLE
  )
  LOCATION \FIGARO.$DATA.ZSDADM53.QUSGEI00
  NAME FIGARO_DATA_ZSDADM53_QUSGEI00
  ;
CREATE TRIGGER CAT.SCH.BTR
BEFORE UPDATE ON CAT.SCH.T074T3 REFERENCING OLD AS
```

```
    MYOLDROW, NEW AS MYNEWROW
WHEN (MYNEWROW.D > MYOLDROW.D) SET MYNEWROW.B =
    MYNEWROW.B + MYOLDROW.D;
  ;
```

- This is an example of SHOWDDL on a view.

```
>>CREATE VIEW V1
     AS SELECT keycol, valcol, ssn, salary
     FROM T2
     TRANSPOSE SSN, salary AS valcol
        KEY BY keycol;

>>SHOWDDL V1;

CREATE VIEW CAT.SCH.V1 AS
  SELECT CAT.SCH.T2.KEYCOL, CAT.SCH.T2.VALCOL,
 CAT.SCH.T2.SSN, CAT.SCH.T2.SALARY
    FROM CAT.SCH.T2 TRANSPOSE CAT.SCH.T2.SSN,
 CAT.SCH.T2.SALARY AS
    CAT.SCH.T2.VALCOL KEY BY CAT.SCH.T2.KEYCOL;
```

- This is an example of SHOWDDL on a stored procedure.

```
>>CREATE PROCEDURE CAT.SCH.T110_IO_NN
  (
  IN IN1 NUMERIC(9,3),
  OUT OUT2 NUMERIC(9,3)
  )
  EXTERNAL NAME 't110.T110_io_nn
(java.math.BigDecimal,java.math.BigDecimal[])'

  EXTERNAL PATH '/usr/ned/regress/udr'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  CONTAINS SQL
  NOT DETERMINISTIC
  ISOLATE
  ;

>>showddl procedure T110_IO_NN;

CREATE PROCEDURE CAT.SCH.T110_IO_NN
 (
 IN IN1 NUMERIC(9,3),
 OUT OUT2 NUMERIC(9,3)
 )
 EXTERNAL NAME 't110.T110_io_nn
(java.math.BigDecimal,java.math.BigDecimal[])'
 EXTERNAL PATH '/usr/ned/regress/udr'
 LANGUAGE JAVA
 PARAMETER STYLE JAVA
 CONTAINS SQL
 NOT DETERMINISTIC
 ISOLATE
 ;
```

- This is an example of SHOWDDL that reports the maximum number of result sets the sales.order_summary procedure returns.

```
>>showddl samdbcat.sales.order_summary;

CREATE PROCEDURE SAMDBCAT.SALES.ORDER_SUMMARY
(
  IN ON_OR_AFTER_DATE VARCHAR(20) CHARACTER SET ISO88591
, OUT NUM_ORDERS LARGEINT
)
EXTERNAL NAME 'SPJMethods.orderSummary
(java.lang.String,long[],java.sql.ResultSet[],java.sql.Result
Set[])'
EXTERNAL PATH '/usr/mydir/myclasses'
LOCATION \ALPINE.$SYSTEM.ZSDCR2C6.L1Z7NW00
LANGUAGE JAVA
PARAMETER STYLE JAVA
READS SQL DATA
DYNAMIC RESULT SETS 2
NOT DETERMINISTIC
ISOLATE
;

--- SQL operation complete.
```

- This is an example of SHOWDDL on an SQL/MX table that contains BLOCKSIZE with a value other than the default value.

```
>>showddl cat.sch.t1;

CREATE TABLE CAT.SCH.T1
(
  C1 INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , C2 INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , C3 INT DEFAULT NULL
  , CONSTRAINT CAT.SCH.T1_104871911_0089 PRIMARY KEY (C1 ASC,
C2 ASC) NOT DROPPABLE
  , CONSTRAINT CAT.SCH.T1_104871911_0088
     CHECK (CAT.SCH.T1.C1 IS NOT NULL AND CAT.SCH.T1.C2 IS
NOT NULL) NOT DROPPABLE
)
LOCATION \FIGARO.$DATA.ZSDADM53.QUSGEI00
NAME FIGARO_DATA_ZSDADM53_QUSGEI00
PARTITION
(
  ADD FIRST KEY (1200)
  LOCATION \FIGARO.$DATA.ZSDWWWWW.AADZ1400
, ADD FIRST KEY (1500)
  LOCATION \FIGARO.$DATA.ZSDWWWWW.AADZ1200
  NAME FIGARO_$DATA_ZSDWWWWW_AADZ1200
)
ATTRIBUTES BLOCKSIZE 32768, MAXEXTENTS 600
STORE BY (C1 ASC, C2 ASC);


CREATE UNIQUE INDEX T1_102261179_0004 ON CAT.SCH.T1
```

```
(
C1 ASC
)
LOCATION \FIGARO.$DATA2.ZSDUXXBK.B7VVVW00
NAME FIGARO_DATA2_ZSDQXXBK_B7VVVW00
ATTRIBUTES BLOCKSIZE 32768, MAXEXTENTS 600
;
```

- The following example displays the Sequence Generator attributes for an IDENTITY column:

```
>>showddl cat.sch.t1;
```

```
CREATE TABLE CAT.SCH.T1
  (
    SURROGATE_KEY LARGEINT GENERATED ALWAYS AS IDENTITY
      (START WITH 99 INCREMENT BY 2 MAXVALUE 900 MINVALUE 50 NO
CYCLE)

      LOCATION \NSK.$SYSTEM.ZSDGNCWF.G423GN00
 -- NOT NULL NOT DROPPABLE
  , B                                INT UNSIGNED NO DEFAULT
      -- NOT NULL NOT DROPPABLE
  , CONSTRAINT CAT.SCH.T1_344545289_4325 PRIMARY KEY
(SURROGATE_KEY ASC) NOT
      DROPPABLE
  , CONSTRAINT CAT.SCH.T1_482894289_4325 CHECK
(CAT.SCH.T1.SURROGATE_KEY IS NOT
      NULL AND CAT.SCH.T1.B IS NOT NULL) NOT DROPPABLE
  )
  LOCATION \NSK.$DATA.ZSDGNCWF.HWD7GN00
  NAME NSK_DATA_ZSDGNCWF_HWD7GN00
  ATTRIBUTES ALIGNED FORMAT, BLOCKSIZE 4096
  STORE BY (SURROGATE_KEY ASC)
;
```

- This is an example of SHOWDDL on a view for which Similarity Check is enabled:

```
>>showddl cat.sch.v1;
```

```
CREATE VIEW CAT.SCH.V1 ENABLE SIMILARITY CHECK AS SELECT
CAT.SCH.T1.I, CAT.SCH.T1.J from CAT.SCH.T1;
```

- This is an example of SHOWDDL on a view for which Similarity Check is disabled:

```
>>showddl cat.sch.v2;
```

```
CREATE VIEW CAT.SCH.V2 DISABLE SIMILARITY CHECK AS SELECT
CAT.SCH.T2.I, CAT.SCH.T2.J from CAT.SCH.T2;
```

- This is an example of SHOWDDL on a sequence generator created with default attribute values:

```
>>create sequence seq1;
--- SQL operation complete.
>>showddl seq1;
CREATE SEQUENCE CAT.SCH.SEQ1 LARGEINT
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 9223372036854775807
NO CYCLE
LOCATION \HPIDMR5.$DATA07.ZSDFGVS8.Z1S98C00
;
```

SHOWLABEL Command

Considerations for SHOWLABEL
Examples of SHOWLABEL

The SHOWLABEL command displays file-label information for SQL/MX objects. This information includes the object version, physical location, and other characteristics. Supported objects are tables, trigger temporary tables, views, and indexes.

SHOWLABEL is an SQL/MX extension.

```
SHOWLABEL {[namespace ] object-name | location-name }
     [, DETAIL ]

namespace is:
       TABLE
     | INDEX

object-name is: [catalog.][schema.]name

location-name is: [\node.]$volume.subvol.filename
```

namespace

specifies the namespace in which the object name is to be searched. If no namespace is specified, the table namespace is used as the default namespace.

object-name

specifies the ANSI name of a table, worktable, or index. This must be an SQL/MX object name because SQL/MP objects and SQL/MP aliases are not supported. If a catalog name and schema are not specified when using an ANSI name, SHOWLABEL uses the default catalog and schema.

location-name

specifies the Guardian physical location of an SQL/MX object. The location name must be the data fork of an SQL/MX object (files that end in “00” and exist in subvolumes beginning with the letters ZSD). SQL/MP objects are not supported.

node is the name of a node on a NonStop system, $*volume* is the name of a disk volume, *subvol* is the name of a subvolume, and *filename* is the name of a Guardian disk file. If the physical name is not fully qualified, it is expanded by using the current default node, volume, and subvolume.

DETAIL

specifies the display of additional information about:

- Security
- Key columns
- Partitions

- Indexes
- Triggers

Considerations for SHOWLABEL

Every SQL object includes a logical file label to store the object's file attributes and information about its dependent objects. The resource fork is a new file that contains structural descriptions of a table. When an SQL/MX object is created, two physical files are instantiated: the data fork and the resource fork. The data fork is where the user data resides. The resource fork contains structural information, such as the partition map.

- You can use SHOWLABEL only within an MXCI session.
- SHOWLABEL does not support stored procedures, SQL/MP objects, or SQL/MP aliases.
- SHOWLABEL displays all output in the English (ISO88591) character set only.
- SHOWLABEL requires that TMF, NonStop SQL/MX, and MXCI be available and running on the system.

SHOWLABEL Output

This table describes SHOWLABEL output. For actual output values, see the examples that follow.

AnsiName	ANSI name of the object.
AnsiNameSpace	Namespace in which the object exists (TA, IX, and so on).
GuardianName	Physical location name of the object
Version	The high level SQL version that was running when the object was created.
ObjectSchemaVersion	The schema version of the object's schema. It is assigned when the object is created and changes when the schema is upgraded or downgraded.
ObjectFeatureVersion	The feature version that describes features used by the database object. Can change as features are added or removed from an object as the result of DDL or utility operations.
Owner	The name of the object's owner.
RedefTimestamp	Date, time, and Julian timestamp indicating when the object's definition was last modified.
CreationTimeStamp	Date, time, and Julian timestamp indicating when the object was created.

LastModTimestamp	Date, time, and Julian timestamp indicating when the object's data was last modified.
LastOpenTimestamp	Date, time, and Julian timestamp of last open time. NEVER OPENED is generated if the object has never been opened (for example, timestamp=0).
SMDtable	Indicates whether the object is a system metadata (SMD) table. User metadata tables are not SMD tables.
File Organization	The file organization of the object (for example, key-sequenced).
Block Length	The length of a block.
File Code	File code in the range 550 through 565.
AuditCompress	Indicates if compressed audit-checkpoint messages are generated for Disk Process 2 (DP2) files.
ClearOnPurge	Indicates disk erasure when the file is dropped.
Audited	Indicates whether the file is audited. If the value is F, a utility operation is in progress or has failed.
Broken	Indicates whether the broken bit is set.
Buffered	Indicates whether the file is buffered
CrashOpen	The file is in crash-open state.
CrashLabel	The file is in crash-label state.
Corrupt	The file is corrupt (the contents of the file are in question). If the value is T, a utility operation is in progress or has failed.
RollfwdNeeded	Roll forward is needed.
RedoNeeded	The file cannot be opened, and media recover (redo) is needed.
UndoNeeded	The file cannot be opened, and media recover (undo) is needed.
IncompletePartBoundChg	Indicates whether a partition boundary change is in progress. If the value is T, a utility operation is in progress or has failed.
UnreclaimedSpace	Indicates whether discarded blocks need to be cleaned up following a partition boundary change. If the value is T, a utility operation is in progress or has failed.
Primary Extent Size	Size of the primary extent in pages.
Secondary Extent Size	Size of the secondary extents in pages.

Max Extents	Maximum number of extents.
EOF	The relative byte address of the first byte of the next available block.
Extents Allocated	Number of extents currently allocated for the file.
Index Levels	Number of index levels used for index blocks.

SHOWLABEL, DETAIL Output

SHOWLABEL, DETAIL returns the SHOWLABEL output and some additional information. This table describes the additional information that SHOWLABEL, DETAIL provides. For actual output values, see the examples that follow:

Record Expression Label Length	Length of the Record Expression label.
Security Label Length	Length of the Security label.
Key Columns	Column number and order of key columns. If the primary key and store by key are not the same, NonStop SQL/MX displays the SYSKEY after the key columns.
Partitioning Scheme	Scheme used for partitioning (for example, RP).
Low Key	Specifies the first partitioning key value that can be stored in the associated partition. Specifies the lowest value for the partition if the column for the value has an ascending order. Specifies the highest value for the partition if the column has a descending order. This information is displayed only for objects that are range partitioned.
ID	ID of the trigger. The timestamp indicating when the trigger was created.
status	Indicates whether the trigger is enabled or disabled.

Examples of SHOWLABEL

- Use the SHOWLABEL command with an object (ANSI) name:

```
>>showlabel tab1;
==============================================================

GuardianName: \CARNAG.$CHINA.ZSDQHSJZ.RGCX9K00
AnsiName: MOD107.SCH.TAB1
AnsiNameSpace: TA

ObjectSchemaVersion: 1200
ObjectFeatureVersion: 1200
```

```
Owner: QADEV.TEG

RedefTimestamp:    29 Feb, 2004 20:34:18 ( 211944875658652063
)
CreationTimeStamp: 29 Feb, 2004 20:31:24 ( 211944875484237505
)
LastModTimestamp:  29 Feb, 2004 20:31:24 ( 211944875484237505
)
LastOpenTimeStamp: NEVER OPENED

SMDtable: F
File Organization: Key-sequenced
Block Length: 4096
File Code: 550

AuditCompress: T
Audited: T (If F, a Utility operation is in progress or has
failed)
Broken: F
Buffered: T
ClearOnPurge: F
Corrupt: F (If T, a Utility operation is in progress or has
failed)
CrashLabel: F
CrashOpen: F

IncompletePartBoundChg: F (If T, a Utility operation is in
 progress or has failed)
RedoNeeded: F
RollfwdNeeded: F
UndoNeeded: F
UnreclaimedSpace: F (If T, a Utility operation is in progress
or has failed)

Primary Extent Size: 16 Pages
Secondary Extent Size: 64 Pages
Max Extents: 160
Extents Allocated: 0
EOF: 0

Index Levels: 0

=============================================================
```

- Use the SHOWLABEL, DETAIL command with an object name:

```
>>showlabel tab1, detail;
=============================================================

GuardianName: \CARNAG.$CHINA.ZSDQHSJZ.RGCX9K00
AnsiName: MOD107.SCH.TAB1
AnsiNameSpace: TA

ObjectSchemaVersion: 1200
ObjectFeatureVersion: 1200
```

```
Owner: QADEV.TEG

RedefTimestamp:    29 Feb, 2004 20:34:18 ( 211944875658652063
)
CreationTimeStamp: 29 Feb, 2004 20:31:24 ( 211944875484237505
)
LastModTimestamp:  29 Feb, 2004 20:31:24 ( 211944875484237505
)
LastOpenTimeStamp: NEVER OPENED
SecurityTimestamp: 29 Feb, 2004 20:31:24 ( 211944875484152346
)

SMDtable: F
File Organization: Key-sequenced
Block Length: 4096
File Code: 550

AuditCompress: T
Audited: T (If F, a Utility operation is in progress or has
failed)
Broken: F
Buffered: T
ClearOnPurge: F
Corrupt: F (If T, a Utility operation is in progress or has
failed)
CrashLabel: F
CrashOpen: F

IncompletePartBoundChg: F (If T, a Utility operation is in
 progress or has failed)
RedoNeeded: F
RollfwdNeeded: F
UndoNeeded: F
UnreclaimedSpace: F (If T, a Utility operation is in progress
or has failed)

Primary Extent Size: 16 Pages
Secondary Extent Size: 64 Pages
Max Extents: 160
Extents Allocated: 0
EOF: 0

Index Levels: 0

Record Expression Label Length: 9168
Security Label Length: 120

Key Columns: 0 ASC

Partitioning Scheme: RP

Partition Array - 3 partition[s]
Partition[0]: \CARNAG.$CHINA.ZSDQHSJZ.RGCX9K00
  Low Key: ( -2147483648 )
Partition[1]: \CARNAG.$KEMPO.ZSDQHSJZ.N9HX9K00
```

```
  Low Key: ( 20 )
Partition[2]: \CARNAG.$LORI.ZSDQHSJZ.HMNX9K00
  Low Key: ( 40 )

IndexMap Array  - 2 index[es]
Index[0]: \CARNAG.$BLANCA.ZSDQHSJZ.JH8TCM00
  Index columns: 1 ASC , 0 ASC
Index[1]: \CARNAG.$BLANCA.ZSDQHSJZ.W7DJ2M00
  Index columns: 0 ASC , 1 ASC , 0 ASC

Trigger Status Array - 2 trigger[s]
Trigger[0]: trigger created (ID): 211944875578177762
  Status: ENABLED
Trigger[1]: trigger created (ID): 211944875658652063
  Status: ENABLED

=============================================================
```

- Use the SHOWLABEL command with a physical location (Guardian) name:

```
>>showlabel \CARNAG.$CHINA.ZSDQHSJZ.RGCX9K00;
=============================================================

GuardianName: \CARNAG.$CHINA.ZSDQHSJZ.RGCX9K00
AnsiName: MOD107.SCH.TAB1
AnsiNameSpace: TA

ObjectSchemaVersion: 1200
ObjectFeatureVersion: 1200

Owner: QADEV.TEG

RedefTimestamp:    29 Feb, 2004 20:34:18 ( 211944875658652063
)
CreationTimeStamp: 29 Feb, 2004 20:31:24 ( 211944875484237505
)
LastModTimestamp:  29 Feb, 2004 20:31:24 ( 211944875484237505
)
LastOpenTimeStamp: NEVER OPENED

SMDtable: F
File Organization: Key-sequenced
Block Length: 4096
File Code: 550

AuditCompress: T
Audited: T (If F, a Utility operation is in progress or has
failed)
Broken: F
Buffered: T
ClearOnPurge: F
Corrupt: F (If T, a Utility operation is in progress or has
failed)
CrashLabel: F
CrashOpen: F
```

```
IncompletePartBoundChg: F (If T, a Utility operation is in
progress or has failed)
RedoNeeded: F
RollfwdNeeded: F
UndoNeeded: F
UnreclaimedSpace: F (If T, a Utility operation is in progress
or has failed)

Primary Extent Size: 16 Pages
Secondary Extent Size: 64 Pages
Max Extents: 160
Extents Allocated: 0
EOF: 0

Index Levels: 0

=============================================================
```

- Use the SHOWLABEL, DETAIL command with a physical location (Guardian) name:

```
>>showlabel \CARNAG.$CHINA.ZSDQHSJZ.RGCX9K00, detail;
=============================================================

GuardianName: \CARNAG.$CHINA.ZSDQHSJZ.RGCX9K00
AnsiName: MOD107.SCH.TAB1
AnsiNameSpace: TA

ObjectSchemaVersion: 1200
ObjectFeatureVersion: 1200

Owner: QADEV.TEG

RedefTimestamp:    29 Feb, 2004 20:34:18 ( 211944875658652063
)
CreationTimeStamp: 29 Feb, 2004 20:31:24 ( 211944875484237505
)
LastModTimestamp:  29 Feb, 2004 20:31:24 ( 211944875484237505
)
LastOpenTimeStamp: NEVER OPENED
SecurityTimestamp: 29 Feb, 2004 20:31:24 ( 211944875484152346
)

SMDtable: F
File Organization: Key-sequenced
Block Length: 4096
File Code: 550

AuditCompress: T
Audited: T (If F, a Utility operation is in progress or has
failed)
Broken: F
Buffered: T
ClearOnPurge: F
Corrupt: F (If T, a Utility operation is in progress or has
```

```
failed)
CrashLabel: F
CrashOpen: F

IncompletePartBoundChg: F (If T, a Utility operation is in
progress or has failed)
RedoNeeded: F
RollfwdNeeded: F
UndoNeeded: F
UnreclaimedSpace: F (If T, a Utility operation is in progress
or has failed)

Primary Extent Size: 16 Pages
Secondary Extent Size: 64 Pages
Max Extents: 160
Extents Allocated: 0
EOF: 0

Index Levels: 0

Record Expression Label Length: 9168
Security Label Length: 120

Key Columns: 0 ASC

Partitioning Scheme: RP

Partition Array - 3 partition[s]
Partition[0]: \CARNAG.$CHINA.ZSDQHSJZ.RGCX9K00
  Low Key: ( -2147483648 )
Partition[1]: \CARNAG.$KEMPO.ZSDQHSJZ.N9HX9K00
  Low Key: ( 20 )
Partition[2]: \CARNAG.$LORI.ZSDQHSJZ.HMNX9K00
  Low Key: ( 40 )

IndexMap Array  - 2 index[es]
Index[0]: \CARNAG.$BLANCA.ZSDQHSJZ.JH8TCM00
  Index columns: 1 ASC , 0 ASC
Index[1]: \CARNAG.$BLANCA.ZSDQHSJZ.W7DJ2M00
  Index columns: 0 ASC , 1 ASC , 0 ASC

Trigger Status Array - 2 trigger[s]
Trigger[0]: trigger created (ID): 211944875578177762
  Status: ENABLED
Trigger[1]: trigger created (ID): 211944875658652063
  Status: ENABLED

==============================================================


--- SQL operation complete.
```

- Use the SHOWLABEL, DETAIL command with a physical location (Guardian) name on a table with a SYSKEY:

```
>>showlabel $FL0115.ZSDJNQHX.Z91KC400,detail;

==============================================================

        GuardianName: \BERT.$FL0115.ZSDJNQHX.Z91KC400
        AnsiName: TESTCAT.TESTSCH.Y4
        AnsiNameSpace: TA

        ObjectSchemaVersion: 1200
        ObjectFeatureVersion: 1200

        Owner: QADEV.TEG

        RedefTimestamp:    23 Nov, 2004 07:28:40 (
211967983720864326 )
        CreationTimeStamp: 23 Nov, 2004 07:28:45 (
211967983725720055 )
        LastModTimestamp:  23 Nov, 2004 07:28:45 (
211967983725720055 )
        LastOpenTimeStamp: NEVER OPENED
        SecurityTimestamp: 23 Nov, 2004 07:28:40 (
211967983720864326 )

        SMDtable: F
        File Organization: Key-sequenced
        Block Length: 4096
        File Code: 550

        AuditCompress: T
        Audited: T (If F, a Utility operation is in progress
or has failed)
        Broken: F
        Buffered: T
        ClearOnPurge: F
        Corrupt: F (If T, a Utility operation is in progress
or has failed)
        CrashLabel: F
        CrashOpen: F

        IncompletePartBoundChg: F (If T, a Utility operation
is in progress or has failed)
        RedoNeeded: F
        RollfwdNeeded: F
        UndoNeeded: F
        UnreclaimedSpace: F (If T, a Utility operation is in
progress or has failed)

        Primary Extent Size: 16 Pages
        Secondary Extent Size: 64 Pages
        Max Extents: 160
        Extents Allocated: 0
        EOF: 0
```

```
        Index Levels: 0

        Record Expression Label Length: 46656
        Security Label Length: 120

        Key Columns: 2 ASC , 3 ASC , 5 ASC , 6 ASC , 7 ASC ,
8 ASC , 9 ASC , 10 ASC, 0 ASC

        Partitioning Scheme: RP

        Partition Array - 1 partition[s]

        IndexMap Array  - 4 index[es]
        Index[0]: \BERT.$FL0115.ZSDJNQHX.PJLVC400
          Index columns: 11 ASC
        Index[1]: \BERT.$FL0115.ZSDJNQHX.CL17H400
          Index columns: 15 ASC
        Index[2]: \BERT.$FL0115.ZSDJNQHX.L3ZHK400
          Index columns: 16 ASC
        Index[3]: \BERT.$FL0115.ZSDJNQHX.VTXSL400
          Index columns: 2 ASC , 3 ASC , 5 ASC

        Trigger Status Array - 0 trigger[s]
```

SHOWSHAPE Command

The SHOWSHAPE command displays the control query shape for a given DML statement. You can use the result at a later time to force the same access plan for the statement. See CONTROL QUERY SHAPE Statement on page 2-62 and SET SHOWSHAPE Command on page 4-66.

Use SHOWSHAPE only within an MXCI session.

```
SHOWSHAPE statement
```

statement

is an SQL DML statement.

Considerations for SHOWSHAPE

Default Control Query Shape

You can use the SHOWSHAPE command for any SQL statement. For those statements that do not have a shape—for example, the CREATE SCHEMA statement—a control query shape (CQS) of the form CONTROL QUERY SHAPE ANYTHING is issued.

CONTROL QUERY SHAPE ANYTHING resets the effect of any preceding CQSs. Its use is especially important when CQSs are being generated from an input file of commands and statements. See SET SHOWSHAPE Command on page 4-66.

Examples of SHOWSHAPE

- Display the access plan for the given statement:

```
SHOWSHAPE
  SELECT E.last_name, J.jobdesc
  FROM persnl.employee E, persnl.job J
  WHERE E.salary > 40000.00
    AND E.jobcode = J.jobcode;

control query shape
  nested_join(partition_access(scan(
  'J', forward, mdam off)),
  materialize(partition_access(scan(
  'E', forward, mdam off))));

--- SQL operation complete.
```

- Display the access plan for the given statement:

```
SET NAMETYPE NSK;
SET MPLOC $DATA06.PERSNL;

SHOWSHAPE
SELECT first_name, last_name, deptnum, salary
FROM persnl.employee
WHERE salary >
  (SELECT MAX (salary)
   FROM persnl.employee
   WHERE deptnum = 1500);

control query shape
  hybrid_hash_join(partition_access(
  sort_groupby(scan(path 'EMPLOYEE',
    forward, mdam off))),
  partition_access(scan(path 'EMPLOYEE',
    forward, mdam off)));

--- SQL operation complete.
```

- This example shows the output of the SHOWSHAPE command using the MultiUnion operator:

```
>>showshape select * from t1 union all select * from t1 union
all select * from t1;

control query shape
expr(MultiUnion(partition_access(scan(path 'CAT.SCH.T1',
forward, blocks_per_access 1 , mdam off)),
partition_access(scan(path 'CAT.SCH.T1', forward,
blocks_per_access 1, mdam off)),partition_access(scan(path
'CAT.SCH.T1', forward, blocks_per_access 1 , mdam off))));
--- SQL operation complete.
```

SHOWSTATS Command

The SHOWSTATS command retrieves statistics from a column(s) of a table. You can use MXCI or an embedded program to execute this command.

```
SHOWSTATS FOR TABLE tablename ON column-spec [DETAIL]

column-spec is:
                        | EVERY KEY [columnname,columnname...]
                        | EVERY COLUMN
                        | EXISTING COLUMN[S]
                        | column1 TO column2
                        | column-set

column-set is:
         column-set-member [, column-set-member...]

column-set-member is:
          columnname | multi-column-group

multi-column-group is: (columnname1, columnname2...)
```

`DETAIL`

displays the detailed statistics, which includes the histogram ID, number of intervals, total number of rows, total UEC, column names, and low and high values for each interval.

`EVERY KEY`

displays the statistics for every column that is a key or a part of the key.

`EVERY COLUMN`

displays the statistics for every column in the table for which statistics are available.

`EXISTING COLUMN[S]`

displays the statistics for every single column and multi-column group of the table for which statistics are available.

`column1` `TO` *`column2`*

displays the available statistics from *`column1`* to *`column2`* in the table .

Consideration for SHOWSTATS

If you have upgraded SQL/MX from an earlier version to 3.2 or later, HP recommends that you run the UPDATE STATISTICS command before executing the SHOWSTATS command. Otherwise, the display order for the column's histogram might differ.

Examples of SHOWSTATS

- This example displays the histogram for every column in the table for which statistics are available:

```
>>SHOWSTATS FOR TABLE student ON EVERY COLUMN;

Histogram data for Table SHOWSTATS.SCH.STUDENT
Table ID: 520295568744812

Hist ID #    Ints    Rowcount         UEC   Colname(s)
=======      ====    ========        ====   ==========
1126276632      6           6           6   ROLLNO
1126276627      6           6           6   NAME
1126276622      6           6           6   DOB
1126276617      6           6           6   SUBJECT
--- SQL operation complete.
```

- This example displays the histogram for the column, `name`:

```
>>SHOWSTATS FOR TABLE student ON name;
Histogram data for Table SHOWSTATS.SCH.STUDENT
Table ID: 520295568744812
Hist ID #    Ints    Rowcount         UEC  Colname(s)
========     =====   =========       ===== ==========
1126276627      6           6           6  NAME
--- SQL operation complete.
```

- This example displays the histogram for the SQL/MP table, `mpteach`:

```
>>SHOWSTATS FOR TABLE \DMR15.$DATA07.CHANVOLM.mpteach ON
EVERY COLUMN;

Histogram data for Table \DMR15.$DATA07.CHANVOLM.MPTEACH
Table ID: 212193205206643373
```

```
Hist ID #   Ints    Rowcount        UEC    Colname(s)
========== ====== ===========       ====   ==========
1013503854      5           5          5   SYSKEY
1013503849      5           5          5   ID
1013503844      5           5          5   NAME
1013503839      4           5          4   SUBJECT
--- SQL operation complete.
```

- This example displays the detailed histogram for the column, dob:

```
>>SHOWSTATS FOR TABLE student ON dob DETAIL;
Detailed Histogram data for Table SHOWSTATS.SCH.STUDENT
Table ID: 520295568744812

Hist ID:    1126276622
Column(s):  DOB
Total Rows: 6
Total UEC:  6
Low Value:  (DATE '1976-05-15')
High Value: (DATE '1999-09-13')
Intervals:  6

Number    Rowcount     UEC       Boundary
======    =========    ====      ==================
  0           0           0      (DATE '1976-05-15')
  1           1           1      (DATE '1976-05-15')
  2           1           1      (DATE '1981-01-22')
  3           1           1      (DATE '1985-12-12')
  4           1           1      (DATE '1987-10-19')
  5           1           1      (DATE '1991-11-25')
  6           1           1      (DATE '1999-09-13')
 --- SQL operation complete.
```

- This example displays the detailed histogram for the multi-column-group (name, dob):

```
>>SHOWSTATS FOR TABLE student ON (name,dob) DETAIL;
   Detailed Histogram data for Table SHOWSTATS.SCH.STUDENT
   Table ID: 1952836866910076444

   Hist ID:    850886066
   Column(s):  NAME, DOB
   Total Rows: 6
   Total UEC:  6
   Low Value:  ('Ajay',DATE '1976-05-15')
   High Value: ('vijay',DATE '1999-09-13')
   Intervals:  1

   Number    Rowcount        UEC    Boundary
   ====== =========== ===========   ========================
     0          0              0    ('Ajay',DATE '1976-05-15')
     1          6              6    ('vijay',DATE '1999-09-13')
   --- SQL operation complete.
```

- This example displays the SHOWSTATS output when a column does not exist:

```
>>SHOWSTATS FOR TABLE student ON nonexistencecolumn;
*** ERROR[9209] Column NONEXISTENCECOLUMN does not exist in
object SHOWSTATS.SCH.STUDENT.
--- SQL operation failed with errors.
```

- This example displays the SHOWSTATS output when there are no histograms for the requested columns:

```
>>SHOWSTATS FOR TABLE student ON name;

Histogram data for Table SHOWSTATS.SCH.STUDENT
Table ID: 520295568744812

Hist ID # Ints    Rowcount         UEC Colname(s)
========  =====   =========       ===== ===========
No Histograms exist for the requested columns or groups
--- SQL operation complete.
```

- This example displays the SHOWSTATS output if there are no histograms for some of the requested columns (histogram for the column `name` is not available):

```
>>SHOWSTATS FOR TABLE student ON name,subject;


*** WARNING[9225] No histogram data is available for columns
(NAME).

 Histogram data for Table SHOWSTATS.SCH.STUDENT
 Table ID: 520295568744812

 Hist ID # Ints    Rowcount         UEC  Colname(s)
 ========= ======  =========       ===== ==========
 1126276641     6          6           6 SUBJECT
 --- SQL operation complete.
```

5 SQL/MX Utilities

A utility is a tool that runs within NonStop SQL/MX or from the OSS shell and performs such tasks as importing data, duplicating files, fixing database discrepancies, and migrating metadata. SQL/MX utilities can be run from MXCI or from the OSS command line.

For a description of MXCI, see MXCI SQL/MX Conversational Interface on page 1-2. For more information about OSS, see the *Open System Services User's Guide*.

For descriptions of the BACKUP and RESTORE commands, see the *Guardian Disk and Tape Utilities Reference Manual*.

This section describes these SQL/MX utilities:

CLEANUP Operation on page 5-3	Removes damaged objects and orphaned Guardian files from an SQL/MX database.
FIXRCB Operation on page 5-7	Performs RCB fixup for all required database objects in catalogs that have an automatic reference on the local system.
FIXUP Operation on page 5-8	Fixes problems in the database and file labels associated with an SQL/MX object.
GOAWAY Operation on page 5-13	Removes Guardian files associated with SQL/MX objects.
import Utility on page 5-18	Imports data from a file into an SQL/MX table.
INFO Operation on page 5-53	Displays file information for Guardian files associated with SQL/MX objects.
mxexportddl Utility on page 5-55	Exports DDL and statistics information to an XML file.
MXGNAMES Utility on page 5-59	Converts ANSI table names into a list of corresponding Guardian file names formatted for TMF or BACKUP/RESTORE 2.
mximportddl Utility on page 5-67	Imports DDL and statistics information from an XML file.
MXRPM tool on page 5-75	Reprocesses module files.
mxtool Utility on page 5-78	Performs FIXUP, GOAWAY, CLEANUP, INFO or VERIFY operations.
VERIFY Operation on page 5-79	Reports whether SQL/MX objects and programs are consistently described in file labels, resource forks, and metadata.

Privileges Required to Execute Utilities

Utility	**Privileges Required**
CLEANUP	Object owner or Super ID if the operation is by ANSI name.
	If the `-oo` option is specified, Super ID is required.
FIXUP operation	Super ID.
GOAWAY operation	Super ID.
import	Have ALL privileges.
INFO operation	Any privilege level.
mxexportddl	Super ID.
mximportddl	Super ID.
MXGNAMES	Have READ/WRITE access to the Guardian subvolume where you are executing MXGNAMES.
VERIFY operation	SELECT privilege on all columns of the table.

CLEANUP Operation

CLEANUP is an OSS command-line utility, run from mxtool, which can remove damaged objects and orphaned Guardian files when a regular DROP operation fails.

The CLEANUP operation supports the following two modes:

- CLEANUP of a single damaged object, by object type and ANSI name.
- CLEANUP of one or more orphaned partitions. An orphaned partition is a Guardian file which represents an SQL/MX partition that is not associated with any object in metadata.

Note. The CLEANUP operation is available only on systems running J06.14 and later J-series RVUs, and H06.25 and later H-series RVUs.

```
mxtool  CLEANUP { object-option | orphaned-files-opt }

object-option is: object-type object-name options

object-type is:
{ TABLE | VIEW | INDEX | SEQUENCE | PROCEDURE  }

object-name is: catalog.schema.object.

options is: [[ -c | -r ][-o=outinfo]]

outinfo is: output-file-name [ CLEAR ]

output-file-name is: an OSS file name

orphaned-files-opt is:
-oo { -f=input-filename  | guardian-physical-filename }[ -
o=outinfo ]

input-filename is: an OSS file name generated by VERIFY
operation, which lists the orphaned files in the node.

guardian-physical-filename is:
[\node.]$volume.subvol.filename
```

object-type

specifies the damaged object for cleanup, and is either TABLE, VIEW, INDEX, SEQUENCE, or PROCEDURE.

object-name

is the ANSI name of the cleanup object. The name must be fully qualified with the catalog and schema.

options

determines how the CLEANUP operation is performed.

`-r` restricts the CLEANUP operation for the specified object. If the object has dependent objects, the CLEANUP operation fails with an error. If no *options* are specified, the default is `-r`.

`-c` cascades the CLEANUP operation for dependent objects. The dependent objects are cleaned up along with the specified object.

input-filename

is the OSS file name of a file generated as a result of running the mxtool VERIFY operation with the `-oo` option. This file contains the Guardian file names of orphaned partitions to be removed.

guardian-physical-filename

is a single Guardian file name of a single orphaned cleanup partition. The node name part is optional.

output-filename

is the OSS path name of the file to which the utility writes the log output. The CLEAR option clears the file before logging the output. If CLEAR is omitted, and if the log file exists, output is appended to the existing data in the file. The *output-filename* cannot have a , (comma) or ; (semicolon).

Considerations

- The CLEANUP operation for a single object can be performed by the schema owner, object owner, or Super ID. The cleanup of orphaned objects can only be performed by the Super ID.
- The CLEANUP operation is used when standard DROP requests fail because of metadata inconsistencies. The CLEANUP operation removes the objects when:
  - The metadata for the object exists and is available, but one or more partitions are physically missing (dangling partition references)
  - One or more partitions physically exist, but the metadata for the object is missing (orphaned partitions)
- The CLEANUP operation creates a log file with details of the outcome. If *outinfo* is not specified, a log file with a name in the format of *LOG_weekday_month_day of month_time_year* is created in the current working directory.
- The cleanup of an object with dangling partition references requires the following to be present in the metadata:
  - Catalog information for the object’s catalog in the CATSYS and CAT_REFERENCES tables in the system schema

- Schema information for the object's schema in the SCHEMATA table in the system schema
- Object information in the OBJECTS table in the DEFINITION_SCHEMA_VERSION_nnnn schema in the object's catalog

- The cleanup of orphaned partitions requires the metadata to be available to determine if the partitions are orphaned. For each partition to be cleaned up, the following information must be available:
  - Catalog information for the partition's catalog must be present in the CATSYS and CAT_REFERENCES tables in the system schema. If no information exists for the partition's catalog, then the partition is considered orphaned.
  - If catalog information for the partition's catalog exists, then schema information for the partition's schema must be present in the SCHEMATA table in the system schema. If no information exists for the partition's schema, then the partition is considered orphaned.
  - If catalog and schema information for the partition's catalog and schema exists, then object information must be present in the OBJECTS table in the DEFINITION_SCHEMA_VERSION_nnnn schema for the partition's catalog. If no information exists for the partition's object, then the partition is considered orphaned.
  - If one or more of the metadata tables mentioned above are not available at the time of cleanup, then it cannot be determined if the partition is orphaned. In this case, the partition is not cleaned up.

Restrictions

- You cannot perform the CLEANUP operation on a schema or a catalog.
- You cannot perform the CLEANUP operation to remove a system metadata object by object name. However, you can remove orphaned partitions from system metadata objects.

Examples

The following example performs cleanup of a table with a dangling partition reference. The log file name is not specified and the CLEANUP operation generates a default log file, `LOG_Wed_Jun_27_00_35_20_2012`.

```
>> mxtool cleanup catalog.schema.sometable
 Hewlett-Packard NonStop(TM) SQL/MX MXTOOL Utility 3.2
(c) Copyright 2003, 2004-2012 Hewlett-Packard Development Company, LP.

Log file containing results of the operation: LOG_Wed_Jun_27_00_35_20_2012

cat LOG_Wed_Jun_27_00_35_20_2012

*************** Time: <time> Process: <process>  Log opened ***************
Cleanup of Table CAT.SCH.SOMETABLE -- Operation Started.
```

```
\COBOLT.$DATA01.ZSDR4XG7.FG5BNL01: error returned 11
Table: CAT.SCH.SOMETABLE
Cleanup of Table CAT.SCH.SOMETABLE -- Operation Finished.
```

Note. The date-time prefix of each log line has been omitted for brevity.

FIXRCB Operation

[Error Conditions](#)
[Example of FIXRCB Operation](#)

FIXRCB is an OSS command-line utility run from `mxtool`. It performs a Record Control Block (RCB) fixup for all the required database objects in catalogs that have metadata tables on the local system. The command must be executed by the Super ID.

```
mxtool fixrcb
```

Error Conditions

One of the following error conditions might occur while executing the `mxtool fixrcb` command:

- An involved node has an incompatible version.
- A user other than the local super ID performed the operation.

Note. The FIXRCB operation does not fix the objects in catalogs that do not have metadata tables on the local system. RCB fixup for objects in such catalogs must be performed on the system where the metadata tables are located.

The FIXRCB operation is available on systems running J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, and on fallback SPR (H06.21-ANC).

Example of FIXRCB Operation

The following command performs an RCB fixup for all the required database objects in catalogs that have an automatic reference on the local system.

```
mxtool fixrcb

Hewlett-Packard NonStop(TM) SQL/MX MXTOOL Utility 3.0
(c) Copyright 2003, 2004-2010 Hewlett-Packard Development
Company, LP.

*** mxtool fixrcb completed successfully ***
```

FIXUP Operation

Considerations for FIXUP Operation
Examples of FIXUP Operation

FIXUP is an OSS command-line utility run from `mxtool` that repairs problems in the SQL/MX database that cannot be repaired by normal operations.

```
mxtool utility-operation

utility-operation is:

FIXUP {guardian-option | object-option}

guardian-option is

  LABEL guardian-file g-opts

guardian-file is [\node.]$volume.subvolume.filename

g-opts is: { -a= { on|off } | -rb | -rc | -rt | -ru }

object-option is

  object-type  object-name  o-opts

object-type is { TABLE | INDEX | SEQUENCE }

object-name is catalog.schema.object

o-opts is: { -rc | -rt | -ru } [-d]
```

guardian-file

specifies the Guardian file to be changed. It must be fully qualified with the volume and subvolume name.

Because the name contains special characters such as "\" or "$", you must precede these characters with a backslash (\), or you can enclose the entire four-part name in single quotes. For example:

`\\node2.\$data3.sales.mytable` or `'\node2.$data3.sales.mytable'`.

g-opts

are options available for a Guardian file:

- Toggle the AUDIT attribute
- Turn off the broken attribute
- Turn off the corrupt attribute
- Reset the redefinition timestamp
- Fix inconsistent label and metadata object UIDs

`{ -a= { on | off }`

toggles the audit attribute on the label. If the audit attribute is ON and you issue a request to turn it on, FIXUP returns a warning that the audit attribute is already on. If the audit attribute is OFF and you issue a request to turn it off, FIXUP returns a warning.

You must be the super ID to perform this operation.

If you turn off auditing for the table, this invalidates online dumps. After the FIXUP operation completes, you must perform a new TMF online dump for all partitions of the table.

`-rb`

turns off the broken attribute on the label. If the broken attribute is already reset and you issue a request to reset it, FIXUP returns a warning.

Caution. The -rb operation to reset the broken attribute on the label is potentially a risky operation. If the file is really broken and you reset the attribute, the consistency of the database will be in question. In addition, the next time DP2 attempts to access the broken file, it resets this attribute.

`-rc`

turns off the corrupt attribute in the label and in the PARTITIONS metadata table for the specific partition. If the partition already has the corrupt attribute turned off and you issue a request to turn it off, FIXUP returns a warning.

Caution. The -rc operation to turn off the corrupt attribute on the label is potentially a risky operation. If the file is really corrupted and you reset the attribute, the consistency of the database will be in question.

`-rt`

sets the redefinition timestamp in label to the value in the OBJECTS table. If the partition already has the correct redefinition time and you issue a request to reset it, FIXUP returns a warning.

You must be the owner of the object or super ID to execute this request.

`-ru`

sets the UID value in the resource fork to match the UID value in metadata. NonStop SQL/MX replaces the catalog, schema, and object UIDs in the resource fork with the values found in metadata.

`object-type`

is an SQL table, sequence generator, or index which has an associated Guardian file.

object-name

specifies the SQL object to be changed. It must be fully qualified with the catalog and schema name.

o-opts

are options available for an SQL object:

- Turn off the corrupt attribute
- Reset the redefinition timestamp

`{ -rc [-d] }`

turns off the corrupt attribute on the label and in the PARTITIONS metadata table.

FIXUP attempts to reset all local partitions associated with the object. If the object has partitions on remote nodes, FIXUP displays a warning and continues.

If an update to the label or metadata fails, the operation fails. If one of the partitions already has the corrupt attribute turned off, FIXUP continues to reset partitions that need to be reset.

If you specify the `-d` option, FIXUP resets the corrupt attribute of all dependencies associated with the object. Table dependencies include indexes and trigger temporary tables. Indexes have no dependencies, so the `-d` option is ignored.

You must be the super ID to execute this request. Both remote and local partitions can have their corrupt attribute reset.

If the metadata and label attributes do not match, FIXUP turns both values off.

△ **Caution.** The -rc operation to turn off the corrupt attribute on the label is potentially a risky operation. If the file is really corrupted and you reset the attribute, the consistency of the database will be in question.

`{ -rt [-d] }`

sets the redefinition timestamp of one or all local partitions of an object to the value saved in the OBJECTS metadata table. If FIXUP cannot extract the redefinition time from the metadata, it returns an error.

FIXUP updates all partitions that make up the object, on the local node.

If the partition has the same timestamp as the metadata, FIXUP continues to update timestamps for partitions that need to be updated. If an update to the label fails, the operation fails.

`{ -ru [-d] }`

sets the UID value in the resource fork to match the UID value in metadata. NonStop SQL/MX replaces the catalog, schema, and object UIDs in the resource fork with the values found in metadata.

If you request the `-d` option, FIXUP updates timestamps for all dependencies associated with the object. Table dependencies include indexes and trigger temporary tables. Each dependent object has its own redefinition timestamp, so each object is set to its own individual time.

You must be the super ID to execute this request.

If the metadata and label timestamps do not match, no warning is issued, but the label value is set to the metadata value.

Considerations for FIXUP Operation

If you change the redefinition timestamp of the label, all executor opens are invalidated. The next time the executor tries to open the file, a similarity check is performed. If it fails, programs are recompiled. The redefinition timestamp is updated whenever the corrupt attribute, broken attribute, or audit attribute is changed.

Examples of FIXUP Operation

Suppose that you create a table, `FIXUPtable`, located in catalog `mycat` and schema `mysch`. It has three partitions, two of which are located on the local node `\local` and one located on a remote node `\remote`. It contains a trigger which requires a trigger temporary table. That table exists on the local node. There are two indexes associated with the table, `index1` and `index2`. Each index has three partitions, organized like the table.

- Suppose a partition on `FIXUPtable` is broken. To fix the problem, you need to turn off the audit attribute, fix the problem, reset the broken attribute, and turn audit back on.

  Partition \LOCAL2.$DATA02.ZSDQ123U.SUEIFO00 is marked broken. Run FIXUP to turn off the audit attribute:

  ```
  mxtool FIXUP LABEL \LOCAL2.$DATA02.ZSDQ123U.SUEIFO00 -a=off
  ```

  Determine the problem and fix it. Then run FIXUP to reset the broken attribute and to turn the audit attribute back on:

  ```
  mxtool FIXUP LABEL \LOCAL2.$DATA02.ZSDQ123U.SUEIFO00 -rb
  mxtool FIXUP LABEL \LOCAL2.$DATA02.ZSDQ123U.SUEIFO00 -a=on
  ```

  Suppose that several of the timestamps on `FIXUPtable` do not match the value on the label. Run FIXUP to reset the timestamps:

  ```
  mxtool FIXUP TABLE mycat.mysch.FIXUPtable -rt -d
  ```

- Suppose a disk or node failure occurs and you must recover table `FIXUPtable`. Following instructions in the *SQL/MX Installation and Management Guide*, you use saved DDL information to recreate the table and recover the privileges in the metadata, then execute the TMF RECOVER FILES command to recover the label, data, and resource forks.

The metadata will now have a new object UID and the label information will have the old object UID. When you perform a VERIFY on this file, because the UID value in the metadata tables does not match the UID value in the resource fork., you receive this message:

```
20799 The { catalog | schema | object } UID in the resource
fork (value) does not match the UID (value) in the metadata
for Guardian file (filename).
```

Run FIXUP to make the UID value in the resource fork match the UID value in the metadata:

```
mxtool FIXUP TABLE mycat.mysch.FIXUPtable -ru -d
```

Run FIXUP to make the UID value in the resource fork match the UID value in the metadata for a sequence generator:

```
mxtool FIXUP SEQUENCE mycat.mysch.seq1 -ru -d
```

GOAWAY Operation

Considerations for GOAWAY
Examples of GOAWAY

GOAWAY is an OSS command-line utility run from `mxtool` that removes Guardian files associated with SQL/MX objects.

SQL/MX files consist of two physical Guardian files, the data fork and the resource fork. Normally, when the data fork is dropped, DP2 automatically drops the corresponding resource fork. In some cases, either an orphaned resource fork or data fork might exist.

GOAWAY does not remove corresponding metadata entries and does not use ANSI names.

```
GOAWAY guardian-file [{-df | -rf | -both}] [-s] [!]

guardian-file is

 [\node.]volume.subvolume.filename
```

Syntax Description of GOAWAY

guardian-file

specifies the Guardian file(s) to be removed. If *guardian-file* is not an SQL/MX object, GOAWAY returns an error. You must fully qualify the file name with the volume and subvolume names. ANSI names are not permitted.

In this four-part name, *node* is the name of a node of a NonStop server, *$volume* is the name of a disk volume, *subvol* is the name of a subvolume that begins with the letters ZSD, and *filename* is the automatically generated name of a Guardian table that ends with "00" or "01" (zero zero or zero one).

If you do not specify *\node*, the default is the Guardian system named in your =_DEFAULTS define. If you specify *\node*, it must be the local system or GOAWAY will return an error. GOAWAY does not drop labels on remote systems.

You can use the "*" Guardian wild card in the volume, subvolume, and file name.

If *guardian-file* does not exist or is inaccessible, an error is returned.

If the Guardian file system encounters an error while searching for files, GOAWAY returns an error. If *guardian-file* is locked, GOAWAY tries for 90 seconds, and then returns a timeout message.

`{-df | -rf | -both}`

directs GOAWAY to drop a single Guardian file (either a data fork or a resource fork), or both.

If you do not specify this option, the default is `-both`.

`-df`

directs GOAWAY to drop a data fork file.

`guardian-file` must be the name of the data fork. Successful delete of a data fork with the data fork option generates an informational message. If you specify a resource fork name, the operation fails.

`-rf`

directs GOAWAY to drop a resource fork file.

`guardian-file` must be the name of the resource fork. If you specify a data fork name, the operation fails.

`-both`

directs GOAWAY to drop both files.

`guardian-file` must be the name of the data fork. If you specify a resource fork name, the operation fails.

If you specify `-both` and only the resource fork exists or only the data fork exists, the operation fails.

If you request the data fork option and a resource fork exists, an error is returned. If you request the resource fork option and a data fork exists, an error is returned.

Valid examples are:

```
mxtool GOAWAY \\figaro.\$vol.subvol.file00 -df

mxtool GOAWAY \\figaro.\$vol.subvol.file01 -rf

mxtool GOAWAY \\figaro.\$vol.subvol.file00

mxtool GOAWAY \\figaro.\$vol.subvol.file00 -both
```

If GOAWAY fails to remove the specified files, it returns an error. Because GOAWAY can do no more, you should notify your service provider.

`-s`

When you run `mxtool`, specific SQL/MX information is extracted including the ANSI name associated with the physical file. This takes some time to process and errors can occur while NonStop SQL/MX is extracting this information. This option allows `mxtool` to skip this step and drop the label.

`!`

When a drop is performed, GOAWAY requests confirmation with each file that matches the list of files specified in *`guardian-file`* to be dropped (data and resource fork). If you specify `!`, GOAWAY does not ask for confirmation but performs the requested operation.

If you do not specify this option, GOAWAY returns with the name of the file it plans to drop. You must confirm (YES) or reject (NO) this action. If you specify YES, the operation continues to the next file that matches the list of files specified in the *guardian-file*. If you specify NO, the operation is aborted on the particular file, a message is generated, and the operation proceeds to the next file that matches the list of files specified in *guardian-file*.

Considerations for GOAWAY

You should use GOAWAY only when no other method of getting rid of an object works.

If you use wild card options and GOAWAY fails after removing one or more files, these files are not rolled back. They are permanently deleted.

Note. When the GOAWAY operation is completed, you must manually change the metadata tables to remove the associated metadata, with a licensed MXCI process.

For more information about using GOAWAY, see the *SQL/MX Installation and Management Guide*.

Examples of GOAWAY

- Confirm a request to drop a file:

```
mxtool goaway \$DATA09.ZSDLLS6G.ZDFXPR00
  ***WARNING[20456] The following file will be removed
  \FIGARO.$DATA09.ZSDLLS6G.ZDFXPR00. Are you sure? (ENTER YES
or NO):
  yes
  Goaway of file: \FIGARO.$DATA09.ZSDLLS6G.ZDFXPR00
successful
  Goaway of file: \FIGARO.$DATA09.ZSDLLS6G.ZDFXPR01
successful
  Ansi Name: CAT.SCH.MVS USED UMD
  Ansi NameSpace: TA
  Object Schema Version: 1200
```

- Drop a file without confirming the drop:

```
mxtool goaway \$DATA09.ZSDLLS6G.WKPRKR00 !
  Goaway of file: \FIGARO.$DATA09.ZSDLLS6G.WKPRKR00
successful
  Goaway of file: \FIGARO.$DATA09.ZSDLLS6G.WKPRKR01
successful
  ANSI Name: CAT.SCH.HISTOGRAM_INTERVALS
  Ansi NameSpace: TA
  Object Schema Version: 1200
```

- Drop a resource fork file:

```
mxtool goaway \$data09.ZSDT6TG2.BLJ35501 -rf !
  Goaway of file:\FIGARO.$DATA09.ZSDT6TG2.BLJ35501 successful
```

- Drop a file and skip the step that extracts ANSI information:

```
mxtool goaway \\FIGARO.$DATA09.ZSDLLS6G.QJ1QNR00 -s !
  Goaway of file: \FIGARO.$DATA09.ZSDLLS6G.QJ1QNR00
successful
  Goaway of file: \FIGARO.$DATA09.ZSDLLS6G.QJ1QNR01
successful
```

- Attempt to drop a file with an invalid volume name:

```
mxtool goaway \DATA09.ZSDKJWZP.BG4LMN00
  *** ERROR[20350] A syntax error was found near 'DATA09',
near character position 7.
```

- Attempt to drop a file without super ID privileges:

```
mxtool goaway \$data09.ZSDT6TG2.HSNZ4500
  *** ERROR[20354] Only super ID can use the MXTOOL operation
GOAWAY.
```

- Attempt to drop a nonexistent file:

```
mxtool goaway \$DATA09.ZSDKJWZP.NONEXI00
  *** ERROR[20355] File $DATA09.ZSDKJWZP.NONEXI00 was not
found.
```

- Attempt to drop an invalid object:

```
mxtool goaway \$system.system.mxcmp
  *** ERROR[20357] File \FIGARO.$SYSTEM.SYSTEM.MXCMP is not
an SQL/MX object.
```

- Attempt to drop a file with an invalid volume specification:

```
mxtool goaway \$vol.ZSDKJWZP.BG4QTN00
  *** ERROR[20362] Error 14 was returned while validating the
file set list specified by '$VOL.ZSDKJWZP.BG4QTN00'.
```

- Attempt to perform GOAWAY with an invalid flag:

```
mxtool goaway \$data09.ZSDT6TG2.HSNZ4500 ! +L=\$data09.zsd0
  *** ERROR[20370] Invalid flag provided for the operation.
Try 'mxtool help'for the operation.
```

- Drop only a data fork:

```
mxtool goaway \$data09.ZSDT6TG2.HMVV3500 -df !
  Goaway of file: \FIGARO.$DATA09.ZSDT6TG2.HMVV3500
successful
  *** WARNING[20372] The resource fork is not accessible.
```

- Attempt to drop only a data fork of a file with both forks:

```
mxtool goaway \$DATA09.ZSDLLS6G.WKPRKR00 -df
  *** ERROR[20450] You asked to drop only the data fork
```

```
  \FIGARO.$DATA09.ZSDLLS6G.WKPRKR00 but a resource fork
exists \FIGARO.$DATA09.ZSDLLS6G.WKPRKR01.
```

- Attempt to drop only a resource fork of a file with both forks:

```
mxtool goaway \$DATA09.ZSDLLS6G.WKPRKR01 -rf
  *** ERROR[20451] You asked to drop only the resource fork
  \FIGARO.$DAT09.ZSDLLS6G.WKPRKR01 but a data fork exists
  \FIGARO.$DAT09.ZSDLLS6G.WKPRKR00.
```

- Attempt to drop a Guardian file on a remote system:

```
mxtool goaway \\SQUAW.\$DATA09.ZSDLLSRG.GJ1GNR00
  *** ERROR[20452] GOAWAY cannot be used to remove a Guardian
file on a remote system \SQUAW.
```

- Attempt to drop a file, but the operation times out:

```
mxtool goaway \$DATA09.ZSDKJWZP.BG4QTN00 !
  *** ERROR[20453] Operation failed on
\FIGARO.$DATA09.ZSDKJWZP.BG4QTN00 due to timeout.
```

- Attempt to drop a file, then abort attempt:

```
mxtool goaway \$DATA09.ZSDKJWZP.BG4QTN00
  *** WARNING[20456] The following file will be removed
  \FIGARO.$DATA09.ZSDKJWZP.BG4QTN00. Are you sure? (Enter YES or NO):

  no
 *** WARNING[20457] GOAWAY aborted at the request of the user.
```

- Attempt to drop a file whose resource fork has been deleted:

```
mxtool goaway \$data09.ZSDT6TG2.HMVV3500
  *** ERROR[20459] You specified -both for the GOAWAY request.
However, only \FIGARO.$DATA09.ZSDT6TG2.HMVV3500 exists. You must use
-df or -rf option to GOAWAY the label.
```

- Attempt to drop both fork files, but specify the resource fork file name:

```
mxtool goaway \$DATA09.ZSDLLS6G.WKPRKR01 -both
  *** ERROR[20460] You specified -both for GOAWAY request of
  \FIGARO.$DATA09.ZSDLLS6G.WKPRKR01. You must specify
  \FIGARO.$DATA09.ZSDLLS6G.WKPRKR00 to GOAWAY the label.
```

- Attempt to drop the data fork, but specify the resource fork file name:

```
mxtool goaway \$DATA09.ZSDKJWZP.BG4QTN01 -df
  *** ERROR[20461] Option -df does not match label
\FIGARO.$DATA09.ZSDKJWZP.BG4ATN01. Use -rf option to GOAWAY the label.
```

- Attempt to drop the resource fork, but specify the data fork file name:

```
mxtool goaway \$DATA09.ZSDKJWZP.BG4QTN00 -rf
  *** ERROR[20462] Option -rf does not match label
\FIGARO.$DATA09.ZSDKJWZP.BG4ATN00. Use -df option to GOAWAY the label.
```

import Utility

The `import` utility imports data from an input file in ASCII or UCS2 format into an SQL/MX table. This utility supports OSS large files (files greater than 2 GB) as input files.

The `import` utility executes at the OSS or MXCI command prompt using the command-line options described next. You cannot directly execute the `import` utility from programs.

Note. You can use DataLoader/MX, in conjunction with `import`, to load and maintain SQL/MP and SQL/MX databases. For more information, see the *DataLoader/MX Reference Manual*.

From an OSS command prompt:

```
import catalog.schema.table  -I input-filename
   [import-option]...

import-option is:
    | -C num-rows
    | -D
    | -E error-filename
    | -F first-row
    | -FD field-delimiter
    | -H help
    | -IP proc-name
    | -L max-errors
    | -LES
    | -PM parsing-errormsg-filename
    | -QL field-qualifier
    | -RD row-delimiter
    | -SF summary-filename
    | -SI summary-interval
    | -T transaction-size
    | -U format-filename
    | -W file-type
    | -XE exec-error-filename
    | -XL max-exec-errors
    | -XM exec-errormsg-filename
    | -Z charset
```

From the MXCI command prompt:

```
sh import catalog.schema.table  -I input-filename
 [import-option]... ;
```

`import`

must be lowercase.

`catalog.schema.table`

specifies the fully qualified name of the destination table for the imported data. You must specify the catalog and schema names.

To import data to the table `mycat.mysch."%&*()"`, you must specify the delimited table name in the `import` command:

```
import 'mycat.mysch."%&*()"' -I myinput.dat
```

The delimited name is enclosed within a pair of double quotes (`"`) and the fully qualified name of the destination table is enclosed in single quotes (‘).

For object names containing single quote ('), use the escape character "\" before special characters and quote. For example, to import data in table `cat.sch."t1'*&"`, use the `import` command:

```
import mycat.mysch.\"t1\'\*\&\" -I myinput.dat
```

`-I` *`input-filename`*

specifies the name of the input file that contains the data to import.

`input-filename` must be an OSS text file (an odd-unstructured file, type 180) or a Guardian text file (type 101). You must specify the file name in OSS format. For example: `/usr/bin/input.txt` or `/G/USER/DATA/INPUT`.

`input-filename` must not contain a minus sign (-) as the first character of the name.

For OSS text file, the newline (\n) character is the line break. For Guardian text file, a carriage return followed by a newline (\r\n) is the line break. You must explicitly mention the row delimiter for Guardian files. For example,

```
import table_name -i /G/USER/DATA/INPUT -RD '\r\n'
```

`-C` *`num-rows`*

is the number of rows (or records) to import. `import` terminates when *`num-rows`* input rows have been imported or when it reaches the end of the input file. If you do not specify this parameter, `import` imports all rows.

`-D`

disables all triggers before the actual insert operation starts and enables the disabled triggers after the import operation is complete.

`-E` *`error-filename`*

specifies the name of the log file for rows in error.

error-filename must be an OSS text file (an odd-unstructured file, type 180) or a Guardian text file (type 101). You must specify the file name in OSS format. For example: `/usr/bin/error.txt` or `/G/USER/DATA/ERROR`.

error-filename must not contain a minus sign (-) as the first character of the name.

-F *first-row*

is the number of the first row (or record) to import. The first row of the input file is designated as the number 0. If you do not specify this number, `import` begins with the first row (which is the same as specifying `-F 0`).

-FD *field-delimiter*

specifies the single or multicharacter field delimiter for the file. The default delimiter is a comma (,). This parameter takes precedence over the field delimiter specified in a format file. To specify a space as a field delimiter, use `" "` or `' '`. If you specify both a field-delimiter and a row-delimiter, they cannot be the same character.

If you are running `import` from the MXCI command prompt and are using special characters as field delimiters, you must enclose special characters in single or double quotes (for example, `'|*|')`.

If you are running `import` from the OSS command prompt and are using special characters as field delimiters, you must use the escape character "\" before special characters.

-H

displays helpful information about `import` command-line options.

-IP *proc-name*

directs `import` to accept data from a DataLoader process, *proc-name*.

-L *max-errors*

directs `import` to ignore the specified number of parsing errors without terminating. Executor errors due to constraint violation are not included in this count. Valid values are 0 through 2147483647.

-LES

is an option to log errors. This option lets you ignore some parsing and execution errors. It also allows the error log filenames and error message log filenames to be automatically generated, instead of specifying values for the new `-XL`, `-XE`, `-XM`, `-PM`, and `-SF` options, and the existing `-L` and `-E` options. The `-LES` option changes the default values for the `-L` and `-XL` options to be 1000 each and the default value for the `-SF` option to be Stdout (For description of the `-SF` option, see section 5.1.2.). Using `-LES` changes the default values for the `-E`, `-XE`, `-PM`, and

-XM options to filenames, which import automatically chooses as unique files in the current directory. Unless the -Z option is used to specify that the input file contains non-ISO88591 characters, import will choose the filenames for the -PM and -XM options similar to the -E and -XE options, respectively, thus intermixing the error messages with the associated rows in error.

If the -LES option is specified and any -L, -XL, -E, -XE, -PM, -XM, or -SF options are also specified, the value(s) specified for the individual options overrides the default value, which the -LES option establishes. The default values established by the -LES option is applicable for any of the seven individual options, which are not specified on the command line.

All error output filenames, including any that are automatically chosen by import, is reported in the results summary file output.

`-PM parsing-errormsg-filename`

is an option that specifies the pathname parsing-errormsg-filename of an OSS output file to which import logs the error messages that correspond with the logged data rows that have parsing errors. The name of the error message file must be specified in OSS format, for example /usr/jdoe/errfile, may be the same as the one specified with the -E option, but must not be the same as any other output file.

This output file is created by the import utility as a non-audited, OSS unstructured file (or a Guardian file of type 180 if the error log file is specified to be under /G). If the file already exists, import terminates with an error message. If the -PM option is not specified, the error messages corresponding to rows with parsing errors are not logged. If the -PM option is specified, but the -E option is not, import terminates with an error message.

If parsing-errormsg-filename and error-filename are specified to be the same file, import logs each error message to the error log file, prefixed by the string ^@ERR_ST]^ and suffixed by the string ^\[ERR_ND^, and by a row delimiter if the input file is a delimited file. The corresponding row in error immediately follows.

If the -Z option is used to specify that the input file contains characters other than ISO88591, and parsing-errormsg-filename and error-filename are specified to be the same file, import will terminate with an error message.

If parsing-errormsg-filename and error-filename are not the same file, the error messages in the error message file will be in the same order as the corresponding logged rows in error in the error file. This is sufficient to allow the user to determine the corresponding error message for any particular row in error-filename.

The error messages written to parsing-errormsg-filename do not have any row numbers to indicate the row in error-filename associated with the error message. However, some parsing error messages do have a row number indicating the associated row in the input file.

Regardless of whether `parsing-errormsg-filename` and `error-filename` are the same file, if you invoke `import` again, with `error-filename` as the input file, the same format file can be used. Any interspersed error messages are automatically ignored by `import`. You must specify the same file type (DELIM or FIXED) when using `error-filename`, as was used for the original input file.

Note. If `-PM` option is specified, you must also consider the space consumed by the error message log file when choosing the `-L` value. Each error message will take at least 80 bytes or more if the error message includes specified column data, and so on.

`-QL` *`field-qualifier`*

specifies a single character field qualifier. The default qualifier is double quote ("). This parameter takes precedence over the field qualifier specified in a format file.

If you are running `import` from an MXCI command prompt and are using special characters as field qualifiers, you must enclose special characters in single or double quotes.

If you are running `import` from an OSS command prompt and are using special characters as field qualifiers, you must use the escape character "\" before special characters.

`-RD` *`row-delimiter`*

specifies the row delimiter *`row-delimiter`*. The default is the end-of-line character (\n). This parameter takes precedence over the row delimiter specified in a format file. If you specify both a field-delimiter and a row-delimiter, they cannot be the same character.

If you are running `import` from an MXCI command prompt and are using special characters as field qualifiers, you must enclose special characters in single or double quotes (for example: `'\r\n'`).

If you are running `import` from an OSS command prompt and are using special characters as row delimiters, you must use the escape character "\" before special characters (for example: `\\r\\n`).

`-SF` *`summary-filename`*

specifies the name `summary-filename` of a non-audited, OSS unstructured file (or Guardian file of type 180 if the filename is under /G) to which `import` writes the number of input rows read so far, the number of input rows that were skipped (either due to the use of the `-F` <first-row> option or because they were comment lines), the elapsed time since starting to read the first input row, and the elapsed time since skipping the initial rows specified via the `-F` <first-row> option. These four lines are repeated after every summary-interval number of rows have been read from the input file. Additional information is appended to the results summary file at the completion of the `import` operation (see Support for restarting import on page 5-49).

The results summary file is created by the `import` utility. If it already exists or if the specified filename is the same as for any error output file, `import` terminates with an appropriate error message. If the `-SF` option is not specified, the summary information will not be generated.

If only `-SF` is specified (without file name), the results summary information is written to Stdout at the completion of the `import` operation. Any positive value specified for the `-SI` option is ignored because the process responsible for updating the four lines of progress information cannot access Stdout for the `import` process.

`-SI` *`summary-interval`*

specifies the number of rows (1 to 2147483647) that `import` has to read from the input file between updates of the information written to the summary file. If this option is not specified, the default configuration writes the results summary information only at the completion of the `import` operation.

Note. It is important to remember that (for best performance reasons and to keep the results summary file from growing very large) the value for summary-interval be at least 10000. The size of the results summary file will increase by approximately 130 bytes each time the four lines of progress information is appended. Therefore, for example, if the input file has 100,000,000 rows and summary-interval is chosen to be 10000, the results summary file will be about 1.3MB in size.

`-T` *`transaction-size`*

specifies the number of records processed before a commit. If `import` returns an error before the record count reaches *`transaction-size`*, the changes to the database within that transaction are rolled back. The number of imported rows is the number successfully committed.

You might want to set the `-T` option to less than 500 to avoid lock escalation. If you do not specify this parameter the default *`transaction-size`* is 10,000 records.

To perform fast load for better performance, `import` turns off the audit attribute on the table before inserting rows. In this case, the `-T` option has no effect because the table is unaudited during the `import` operation.

`-U` *`format-filename`*

specifies the name of an OSS or Guardian text file that contains format specifications for *`input-filename`*. Using a format file is optional.

`format-filename` must be an OSS text file (an odd-unstructured file, type 180) or a Guardian text file (type 101). You must specify the file name in OSS format (for example: `/usr/bin/format.txt` or `/G/USER/DATA/FORMAT`).

`format-filename` must not contain a circumflex (^) character or a comma (,) and must not contain a minus sign (-) as the first character of the name.

`-W` *`file-type`*

specifies the input file type. The possible values for *`file-type`* are DELIM and FIXED. If you do not specify this parameter, `import` assumes the default DELIM file type (a delimited input file). If you do not specify this parameter and there is no format file, `import` assumes the default delimiters for a delimited input file.

If you use the `-W` option, it must follow the `-IP` option on the command line.

`-XE exec-error-filename`

is an option that specifies the `pathname exec-error-filename` of an OSS output file to which `import` logs data rows that have execution errors. After the rows in the error file are edited to resolve the execution errors, the resultant error file can be used as an input file by a subsequent `import` execution.

The name of the error file must be specified in the OSS format, for example `/usr/jdoe/errfile`, and must not be the same file as the one specified with the `-E` error-filename option if the `-E` option is also specified. This output file is created by the `import` utility as a non-audited, OSS unstructured file (or a Guardian file of type 180 if the error log file is specified to be under /G). If the file already exists, `import` terminates with an error message. If the `-XE` option is not specified, rows with execution errors shall not be logged. If the `-XE` option is specified but the `-XL` option is not, `import` terminates with an error message.

Note. In Release 2.0, if you specify a parsing error file using the `-E` option and if the file currently exists, `import` purges the contents of the file before logging the first parsing error into the file. The handling of the parsing error file is changed so that `import` terminates with an error message if the specified parsing error file already exists.

Note. In Release 2.0, if you specify the `-E` option but not the `-L` option, `import` does not report an error, it creates the file specified via the `-E` option as a zero-length file. It does not write any data rows to the file since the default action when no `-L` option is specified is to not accept (and hence not log) any rows with parsing errors. The handling of the `-E` option is changed so that `import` terminates with an error message if the `-E` option is specified without the `-L` option also being specified.

Note. Although the `-XL` option allows a maximum of 2,147,483,646 errors to be ignored, the execution error log file grows to consume a large amount of disk space long before that many errors were actually encountered. Even if the average input row size were only 50 bytes, the error log file grows to the current OSS file size limit of 2,090,950,656 bytes when the number of encountered errors is only 41,819,013. Therefore, it is important to remember that if the `-XE` option is used, the potential size (and disk space consumption) of the error log file must be considered when choosing the `-XL` value.

`-XL max-exec-errors`

is an option that directs `import` to ignore the specified number `max-exec-errors` (from 0 to 2147483646) of ignorable execution errors without terminating. If the number of execution errors detected by `import` exceeds `max-exec-errors`, `import` shall terminate, although because of internal bundling of multiple rows for insertion, `import` may report several more than `max-exec-errors` detected when it does terminate.

If this option is not specified, the default value shall be zero and if any row encounters an execution error, `import` terminates and writes the associated error message to `Stdout` (similar to the current version of `import`). Regardless of the value specified for `max-exec-errors`, non-ignorable execution errors cause `import` to terminate and write the associated error message to `Stdout`.

If `import` terminates before exhausting the input data, any rows that were inserted but not yet committed are backed out (similar to the current `import` utility).

`-XM exec-errormsg-filename`

is an option that specifies the pathname `exec-errormsg-filename` of an OSS output file to which `import` logs the error messages that correspond with the logged data rows that have execution errors. The name of the error message file must be specified in OSS format, for example `/usr/jdoe/errfile`, and may be the same as the one specified with the `-XE` option, but must not be the same as any other output file.

This output file shall be created by the Import utility as a non-audited, OSS unstructured file (or a Guardian file of type 180 if the error log file is specified to be under /G). If the file already exists, `import` terminates with an error message. If the `-XM` option is not specified, the error messages corresponding to rows with execution errors shall not be logged. If the `-XM` option is specified but the `-XE` option is not, `import` terminates with an error message.

If `exec-errormsg-filename` and `exec-error-filename` are specified to be the same file, `import` logs each error message to the error log file, prefixed by the string ^@ERR_ST]^ and suffixed by the string ^\[ERR_ND^ and by a row delimiter if the input file is a delimited file. The corresponding row in error will immediately follow.

If the `-Z` option is used to specify that the input file contains characters other than ISO88591, and `exec-errormsg-filename` and `exec-error-filename` are specified to be the same file, Import will terminate with an error message. (Displaying or editing a file with characters from more than one character set would be very difficult, if not impossible.)

If `exec-errormsg-filename` and `exec-error-filename` are not the same file, the error messages in the error message file will be in the same order as the corresponding logged rows in error in the error file. This should be sufficient to allow the user to determine the corresponding error message for any particular row in `exec-error-filename`.

The error messages written to `exec-errormsg-filename` will not have any row number in them to indicate the row in the input file or in `exec-error-filename` associated with the error message.

Regardless of whether `exec-errormsg-filename` and `exec-error-filename` are the same file, if you invoke `import` again with `exec-error-filename` as the input file, the same format file can be used. Any interspersed error messages will be automatically ignored by `import`. You must specify the same file type (DELIM or FIXED) when using `exec-error-filename` as was used for the original input file.

It is important to note that Dataloader/MX has a pass-through mode where the input data is sent on as the output data without performing any transformations on the data. That capability should allow an error log file produced by `import` (with or without interspersed error message strings) to be used as the input file for Dataloader/MX, if desired.

Note. If the `-XM` option is used, you should also consider the space consumed by the error message log file when choosing the `-XL` value. Each error message will take at least 65 bytes and may be much longer if the error message includes a specified table name, check condition name, and so on. If `exec-errormsg-filename` and `exec-error-filename` are the same file, the error messages also have the indicator prefix and suffix strings and with both the error messages and the rows in error going into the same file, an even smaller `-XL` value may be appropriate.

`-Z` *`charset`*

specifies the character set for the data being imported. Valid values are ISO88591 or UCS2. The default value for `-Z` option is ISO88591. For details about character set conversion, see Data Types of Input Values for Input File on page 5-34.

Considerations for import

Fast Loading and Transaction Considerations

If you are importing into an empty table, `import` uses the fast-loading technique if the target table meets these criteria:

- It is empty.
- It has no indexes.
- It has no droppable primary key, unique key, or foreign key constraints.
- It has no enabled triggers or you specified the `-D` option to disable triggers.

To improve the performance of the fast-loading technique, `import` turns off the audit attribute for the entire table at the start of the operation and turns it back on when the operation ends. If another `import` operation is attempted on the same table while an import operation with the fast-loading technique is being performed, the second operation fails with a concurrent access error.

If you turn off auditing for the table, online dumps are invalidated. After the `import` operation completes, you must perform a new TMF online dump for all partitions of the table. When you specify the transaction size using the `-T` option, transactions are enforced, and the audit attribute of the table is not altered even if the table meets the rest of the criteria for using the fast-loading technique.

DDL Locks

When `import` uses the fast-insert technique, a DDL lock is held on the object for the entire duration of the operation. This strategy prevents any concurrent `import` operation or any DDL or utility operation on this table until the first `import` using the fast-insert technique is complete. If you use multiple import processes to load different partitions of an empty table that meets the described criteria, only the first `import` operation would be able to use the more efficient fast-insert technique. Start the remaining `import` operations after the first one completes and VSBB inserts will be used with TMF transactions

If the import operation fails unexpectedly, you must run the RECOVER utility to remove the DDL lock and perform cleanup. For expected errors, error handling will ensure that the data in the table is purged.

Import records the status in the DDL_LOCKS metadata table. You can query this table to determine the progress of import operation.

Status	Description
1	The DDL Lock row is created.
2	Audit is turned off as part of import.
3	Data copy completed.

Recovery

If `import` fails, you must run the RECOVER utility command to clean up the failed operation.

- If `import` fails to reset the audit attribute of the table that was altered during the operation, specify RECOVER with the `CANCEL` or `RESUME` option to reset the audit attribute.
- If `import` fails after successful data insertion and fails to reset the audit attribute, specify RECOVER with the `RESUME` option.
- If `import` fails before the data is successfully inserted, specify RECOVER with the `CANCEL` option to remove the DDL lock and remove partial data if inserts were not done.

You can find this information by reading the DDL_LOCK table. If you run the RECOVER operation with the incorrect option, RECOVER displays an error message so you can rerun it with the correct option. For details, see Checking DDL Locks on page 2-9.

If the import operation fails unexpectedly, the RECOVER utility does not re-enable triggers that were disabled before running `import`. You must re-enable them. For expected errors, error handling ensures that triggers are re-enabled.

No restart facility is available to handle partially copied data.

Concurrency

If you are importing into an empty table or if the target table does not have index or referential integrity constraints, `import` uses fast-loading techniques. Concurrent import operations are not allowed. Concurrent DML, DDL, and utility operations are also not allowed.

If `import` is not using fast-loading techniques, all DML operations (SELECT, UPDATE, DELETE, INSERT) can be performed concurrently. If too many locks are on the partition, DP2 escalates to a table lock, which prevents concurrent DML operations. Utilities that read only metadata (EXPORTDDL, INFO, MXGNAMES, SHOWDDL, SHOWLABEL, VERIFY) can be performed concurrently.

Parallel imports on the same table are allowed. DDL operations are not recommended.

Format File Sections for import

A format file is optional and, if used, consists of up to four sections. You specify the format file by using the option `-U` *`format-filename`*. A sample format file is provided in the `import` example (see Format File Describing the Data). The sections you can include are:

- [DATE FORMAT]
- [COLUMN FORMAT]

- [DELIMITED FORMAT]
- [FIXED WIDTH FORMAT]

The format file structure and field options that are available are:

Format File Structure	Format File Field Options
`[DATE FORMAT]`	
`DateOrder=`*`order`*	Order of the date fields: MDY: month/day/year DYM: day/year/month YMD: year/month/day YDM: year/day/month DMY: day/month/year MYD: month/year/day Default is MDY.
`DateDelimiter=`*`date_delim`*	Delimiter for the date. Default is a slash (for example, mm/dd/yyyy).
`TimeDelimiter=`*`time_delim`*	Delimiter for the time. Default is a colon (for example, hh:mm:ss).
`FourDigitYear=`*`four_year`*	Y or N. Default is Y; 4 digits (for example, 12/25/1997). You have the option of specifying a two-digit year date or timestamp value. If you specify two digits and the year value is less than 30, `import` assumes the first two digits of the year to be 20. If the year value is greater than or equal to 30, import assumes the first two digits of the year to be 19.
`DecimalSymbol=`*`decimal`*	One character. Default is a decimal point (.). You can specify some character, other than the default decimal point (.) character, to be used whenever the input file has a time value that includes a fractional part of a second. This DecimalSymbol character is used only with a fractional second. It is not recognized in general numeric input, such as 12345.67, or in floating point numbers. For general numerics and floating point numbers, you must use the decimal point.
`NormalizeDate=`*`normalize`*	Y or N. Default is N; no datetime normalization.

`[COLUMN FORMAT]`

`col=`*`field_name,skip`* `[,`*`NullDefault_flag`*`]`	The source field name and whether to skip the field in the source data file. If data is to be stored in the target table, *`field_name`* must be the name of the target column. *`skip`* is Y or N. Default is N: do not skip. *`NullDefault_flag`* is Y or N. Default is N: no null or default flag preceding the data. If the flag in the input field is N, null is inserted into the column. If the flag is D, the default value is inserted into the column. Otherwise, the input field value is inserted. Each field in the source data file has a corresponding `col` entry.

`[DELIMITED FORMAT]`

`FieldDelimiter=`*`field_delim`*	One or more characters that separate fields in a row or record. Default is a comma (,).
`RowDelimiter=`*`row_delim`*	One or more characters that separate rows. Default is the new line character (\n).
`Qualifier=`*`qualifier`*	One character that can enclose a field in a row or record. Default is a double quote (").

`[FIXED WIDTH FORMAT]`

`RecordLength=`*`record_length`*	A decimal ASCII number that specifies the physical record length (in characters, not bytes) of each input row. The length must include the row delimiter, if used. You must specify *`record_length`* for fixed width format. The value specified for *`record_length`* must be greater than or equal to (*`start`* + *`length`* - 1) / (number of bytes per character) for all subsequent col= lines in this section of the format file. All input records must be the same length, exactly *`record_length`* characters long.
`NullValue=`*`null_char`*	A character denoting null. Default is space. If an input field consists of all *`null_char`*, null is stored in the target column.

`col=`*`column_name`*`,`*`start`*`,`*`length`*`[,`*`varcharPrefix_length`*`]`	The target table column name and the start position and length of the input field in each row of the source data file. *start* is a decimal ASCII number that specifies the byte position of the first character of the field (where 1, not 0, refers to the first byte of the input row). *length* is a decimal ASCII number that specifies the number of bytes in the field. For fields whose target is a VARCHAR column, you can optionally specify the actual length of the data as a decimal ASCII number (of characters, not bytes) at the beginning of the data in the input field. If you do, *start* must specify the byte position of the actual length value and all input rows must use the same number of characters at the beginning of the input field to contain the actual length value, and you must specify the *varcharPrefix_length* parameter on the col= line for the column. If you specify *varchar_Prefix_length*, it must be a decimal ASCII number that specifies the number of characters, not bytes, at the beginning of the input field data that are used to contain the actual length value. If the null or default flag *NullDefault_flag* is defined, it precedes the length prefix (if any) in the input field. Each field in the source data file must have a corresponding col= entry. Any of the *record_length* number of characters in each input record that are not covered by a col= entry will be ignored.

Format File Considerations––import

Format File for a DELIM Input File

If the input file type is DELIM and you want to use a format file, you must include the [COLUMN FORMAT] section. The other sections are optional. See [COLUMN FORMAT] on page 5-30.

Format File for a FIXED Input File

If the input file type is FIXED, you must specify a format file that includes the [COLUMN FORMAT] and [FIXED WIDTH FORMAT] sections. The columns listed in the [COLUMN FORMAT] section must match the columns listed in the [FIXED WIDTH FORMAT] section. The other sections are optional. See [FIXED WIDTH FORMAT] on page 5-30.

Input File Considerations––import

Fixed Input File

In a fixed input file, specified by using the `-W FIXED` option, different columns might have different lengths, but for each column, all rows must be the same length. You should pad column values that are shorter than the column width with spaces or NullValue characters to ensure that every row has exactly the same number of bytes for a given column.

Delimited Input File

A delimited input file, specified by using the `-W DELIM` option or by default, uses field and row delimiters and field qualifiers if needed. If you specify a format file, `import` uses the delimiters in the file.

If you do not specify a format file, `import` uses these default delimiters:

Field delimiter	One or more characters used to separate fields in a row or record. The default is a comma (,).
Row or record delimiter	One or more characters used to separate rows or records. The default is the new line character (\n).
Field qualifier	A character used to enclose a field of a row (or record). The default is a double quotation mark(").

This example shows a row from a delimited file with default coding and field qualifiers:

```
"135","Jane Jackson","100 East St.","Cupertino","CA","95014"
```

You are not required to use field qualifiers. In this record, the fields are correctly delimited by a comma(,), and field qualifiers are not needed:

```
135,Jane Jackson,100 East St.,Cupertino,CA,95014
```

Using a Field Qualifier

Use a field qualifier to include field or row delimiters as part of the field data. For example, suppose that your input file uses a comma (,) to delimit the fields in a record. Suppose further that a record contains a field consisting of these characters:

```
Jackson,Jane
```

You can use a field qualifier to ensure that the comma (,) is included in this field. You are not required to use a field qualifier for other fields in the row. For example:

```
135,"Jackson,Jane",100 East St.,Cupertino,CA,95014
```

If your field data contains a default field qualifier of double quote ("), enclose this field data within field qualifiers. For example:

```
135,"Re: "Meeting Request" subject", 01-JUL-1985
```

If the data is enclosed within field qualifiers, HP recommends that you use a unique multicharacter string as a field delimiter that can be distinguished from the enclosed data.

Using a Field Delimiter

Use field delimiters to separate field data for a record. The default field delimiter character is a comma (','). HP recommends that you use a unique multicharacter field delimiter string that is not part of enclosed or nonenclosed field data. For example:

```
135|*| "Re: "Meeting Request|*|" subject" |*| 01-JUL01985
```

In this example, the field delimiter is part of the data. When `import` processes this type of data, as soon as it encounters the first `|*|` in the data that corresponds to the second field, it is treated as a field delimiter. `import` then processes the data following this first `|*|` as next field data, and so on through the data. Therefore, in this example, the column count of the data is considered to be more than the table column count.

Row or Record Delimiters

The new line character (\n) is typically used as a record delimiter in an input data file. If a new line character already exists in an input file as a record delimiter, you cannot specify and include a different record delimiter in the file. If you do, `import` interprets the new line character as part of a data field.

Under some circumstances, you might want to include a new line character as part of a data field. For example, suppose that you have data that is to be used as printed text, and the new line character is included in the data for the purpose of formatting. Then you must specify a record delimiter other than the new line character.

Null Input Values for Delimited Data Input Files

For a delimited input file, if a column in the target table allows null, you can specify null for that column in the input file. Two consecutive field delimiters specify null.

For example, suppose that the EMPLOYEE target table begins with the columns EMPNUM, FIRST_NAME, MIDDLE_INITIAL, and LAST_NAME. The MIDDLE_INITIAL column allows null. Some employees have no middle initial. As a result, the input file contains records like this:

```
2961,Mary,,Smith,143,3490,80000.00
```

To insert null in a nullable target column, you can specify two consecutive field delimiters as shown in the preceding example. To insert blanks in the target column, you can specify two field delimiters with the appropriate number of blanks between the delimiters.

Default Values for Delimited Data Input Files

For a delimited input file, if a column in the target table does not allow null, using two consecutive field delimiters directs `import` to use the default value for the column (instead of null for the column). For example, a column defined as NOT NULL might have space as its default value. In this case, two consecutive field delimiters in the corresponding input field specify that space is to be stored in the target column.

Null Input Values for Fixed-Width Files

For a fixed-width input file, if a column in the target table allows null, you can specify a null indicator character for that column in the input file. See [FIXED WIDTH FORMAT] on page 5-30.

For example, suppose that the EMPLOYEE target table begins with the columns EMPNUM, FIRST_NAME, MIDDLE_INITIAL, and LAST_NAME. Some employees have no middle initial. You specify a hyphen (-) as the null indicator. As a result, the input file contains records like this:

```
296 Mary      -Smith         143 349080000.00
```

Null or Default Flag

For a delimited or fixed-width input file, if a column in the target table allows null or has a default value, you can specify a null or default flag for that column in the input file. The [COLUMN FORMAT] section of the format file indicates that this flag is used in the input file. See `[COLUMN FORMAT]` on page 5-30. For details about importing into nullable columns, see import and Nullable Columns on page 5-37.

These column definitions, for example, allow these insert values:

Column Definition	Insert Value
NOT NULL NO DEFAULT	Input value must be provided. Null or default value is not allowed.
NOT NULL DEFAULT 'abc'	If flag is D, default value is inserted. Otherwise, input value is inserted. Null is not allowed.
NO DEFAULT	If flag is N, null is inserted. Otherwise, input value is inserted. Default value is not allowed.
DEFAULT 'abc'	If flag is D, default value is inserted. If flag is N, null is inserted. Otherwise, input value is inserted.

Two-Digit Year Input Values for Input File

You have the option of specifying a two-digit year date or timestamp value in the [DATE FORMAT] section of a format file. If you do and the year value is less than 30, `import` assumes the first two digits of the year to be 20. If the year value is greater than or equal to 30, `import` assumes the first two digits of the year to be 19.

Data Types of Input Values for Input File

`import` converts the character data in the input file to the appropriate data types as defined in the target table. The data types of the values in an input record must be compatible with the data types of the columns in the destination table.

Use the `-Z` option to specify the character set for the data being imported. `import` will import data in a UCS2 input file to any noncharacter-typed column after a Unicode-to-ISO88591 conversion, and to a character-type column through a conversion that translates the Unicode data to the character set of the column. `import` will directly import UCS2 data to UCS2 columns without conversion.

Format and Data File Requirement for Unicode import

A UCS2 data input file must be in UTF-16BE (UTF-16 big-endian) or UTF-16LE (UTF-16 little-endian) format. The byte order mark (BOM) must occupy the first two bytes of the input file.

Because the format file or the `import` command line is specified in ASCII, the field delimiter, the field qualifier, or the row delimiter character in a UCS2 input data file must be the ASCII-equivalent version. For example, the field delimiter ',' (ASCII value 0x2C) for a Unicode data file must be supplied in Unicode 0x002C.

If the Unicode data file is subjected to a fixed width format importing, the unit for the start position, the length of input fields in the COL attribute, and the record length in the RecordLength field in the FIXED width format section is in characters, not in bytes.

If the Unicode data file is subjected to a fixed width format importing, the unit for these components is characters, not bytes:

- start position
- length of input fields in the COL attribute
- record length in the RecordLength field in the FIXED width format section

Error Reporting for Unicode import

Error messages sent to the console remain in ASCII. If UCS2 data is to be included in the message, its content is converted first. All UCS2 characters in the range [0x00. 0xFF] are converted to an 8-bit ASCII equivalent. For all other UCS2 characters, NonStop SQL/MX uses the hexadecimal form of their code values. `import` inserts a space before and after the hexadecimal value for readability.

Error rows logged to the error log file are in UCS2, in the same byte order as the source data file. You can resubmit the log file to `import` after errors have been corrected.

If a character in the data file cannot be translated into one required by the target column, NonStop SQL/MX issues an error. Importing an UCS2 data file into an ISO88591 column with character' code values beyond the range of ISO88591 leads to this translation error condition.

Datetime and Interval Data for Input File

For datetime and interval data, do not specify keywords that are part of the data type in the column definition.

For example:

- If the datetime value to be imported is `DATE '1990-01-22'`, specify the field without the keyword in the input data file, as:

  ```
  1990-01-22
  ```

- If the interval value to be imported is `INTERVAL '03-04' YEAR TO MONTH`, specify the field without the keywords in the input data file, as:

  `03-04`

Datetime Normalization

NonStop SQL/MX supports the three standard ANSI SQL:1999 datetime formats, which can be loaded without normalization.

Date formats other than ANSI SQL:1999 formats must be normalized to substitute the appropriate year, month, and day delimiters; zero-pad missing digits; and transpose year, month, and day field order. Time formats other than ANSI SQL:1999 formats must be normalized to substitute the appropriate hour, minute, and second delimiters, and zero-pad missing digits. See [DATE FORMAT] on page 5-29.

Transaction Considerations for import

`import` might automatically turn auditing off for all partitions on an empty table without indexes. Auditing is turned back on after the import operation completes or if it fails for any reason. This behavior allows `import` to take advantage of fast loading techniques and to avoid TMF transaction issues. Turning off auditing for the table invalidates online dumps. After the import operation completes, you must perform a new TMF online dump for all partitions of the table.

If multiple import processes are started on different partitions of a table without indexes, the first import operation turns auditing off for all the partitions of the table. In this scenario, only the first import operation would benefit from the efficient insert technique. A warning about performance is issued for other import processes on the same table, because the table can be unaudited and nonempty.

If the table is not empty or has dependent indexes, import continues with the normal load operation using TMF transactions.

import and Nullable Columns

Suppose you need to import into a table that allows nullable columns. Follow these guidelines:

If you specify that the input file is in **delimited** format:

- You can specify a null value for a particular column value with a record in the input file by using two successive field delimiters. In the case of the first column, start the record with a field delimiter. In the case of the last column, end the record with a field delimiter just before the row delimiter.
- If you specify a format file, import ignores the [FIXED WIDTH FORMAT] section, and ignores any NullValue= line.
- In the [COLUMN FORMAT] section of the format file, you could specify Y for the nullDefault_flag portion of the col=field_name,skip,nullDefault_flag line. If you do this, then on any particular input record:
    - You can specify a null value for that column either by specifying two successive field delimiters, or by starting the input data specification with the character “N” or “n”. import ignores any characters after the “N” or “n” and before the next field delimiter or row delimiter.
    - You can specify that the column be given the default value (for the column) by starting the input data specification with the character “D” or “d”. import ignores anything following that and preceding the next field delimiter or row delimiter.
    - If the first character of the input data specification is anything other than “N”, “n”, “D”, or ‘d’, import ignores the first character. import uses data beginning in the next character position up to the next field delimiter or row delimiter as the value for the column.

If you specify that the input file is in **fixed** format:

- You can specify a null value for a particular column value by specifying all of the characters in the fixed-length field as the NullValue character.
- If you specify a format file with a [FIXED WIDTH FORMAT] section that contains a NullValue= line, the specified character is taken as the NullValue character. Otherwise, the NullValue character defaults to the space character (0x20 for ISO88591 input files or 0x0020 for UCS2 files).
- In the [COLUMN FORMAT] section of the format file, you could specify Y for the nullDefault_flag portion of the col=field_name,skip,nullDefault_flag line. If you do this, then on any particular input record:
    - You can specify a null value for that column either by specifying two successive field delimiters, or by starting the input data specification with the character “N” or “n”. import ignores any characters after the '“N” or “n” and before the end of the input field.

    - You can specify that the column be given the default value (for the column) by starting the input data specification with the character “D” or “d”. import ignores anything following that and preceding the end of the input field.
    - If the first character of the input data is anything other than “N”, “n”, “D”, or “d”, import ignores the first character. import uses the data beginning in the next character position up to the end of the input field as the value for the column.

- The col=column_name,start,length line in the format file specifies the byte offset (within each row) and the length (in bytes) where you specify the value for that column. If you specify Y for the nullDefault_flag for a particular column, the leading indicator character (where you might put “N”, “n”, “D”, or “d”) is part of the specification of the value for the column, so the start value should be the byte offset where that leading indicator character is found, and the length value should include the leading indicator character, because it is part of the specification for the value.

Parallel Load for import

Use parallel load when the destination table is partitioned. When using parallel load:

- Some data types require more CPU time during import and therefore parallel load would be a benefit.
- Sorting input data by storage key results in faster import time.
- More processors improve parallel load performance.

You cannot import into one partition in parallel. You receive a locking error if you have two instances of `import` loading the same partition.

You can perform a parallel load in two ways:

- Run multiple instances of `import`—one for each partition in the destination table—to load data into a partitioned table by using a single input file. For each `import` command, specify the number of input rows (or records), the number of the first record to import, and the transaction size. The number of the first record to import begins with zero.

  Note. For better performance, you must specify the transaction size. If you do not, and other import processes are running on the same table, `import` issues a warning regarding performance. If the table is empty and does not have any indexes, the first import process might turn off auditing for all partitions. Turning auditing off enables the first import process to use a fast loading technique. However, performance of the parallel import processes on the remaining partitions is affected, because auditing for all partitions is off, and the table contains data as a result of the first import operation.

  For example, suppose that you partition the EMPLOYEE table into three partitions. The first partition begins with 0 for the employee number, the second partition begins with 3000 for the employee number, and the third partition begins with 5000 for the employee number.

You might specify the three `import` commands as:

```
C:\>import corpcat.persnl.employee -I empfile -C 2999 -T 500
C:\>import corpcat.persnl.employee -I empfile -C 1999 -F 3000
-T 500
C:\>import corpcat.persnl.employee -I empfile -C 1999 -F 5000
-T 500
```

The number of records for each partition must be less than or equal to the space available in each partition, and the rows to be imported into each partition must have an appropriate clustering key. In the preceding example, the first partition allows for employee numbers ranging from 0 to 2999.

Note. Check that the ranges specified are exact (for example, no gaps or omissions and no overlap of rows).

- Run multiple instances of the `import` utility—one for each partition in the destination table—to load data into a partitioned table by using a separate input file for each partition. Each of the input files contains the data for each partition.

  For example, you might specify the three `import` commands for the partitioned EMPLOYEE table as:

```
C:\>import corpcat.persnl.employee -I empfile1 -T 500
C:\>import corpcat.persnl.employee -I empfile2 -T 500
C:\>import corpcat.persnl.employee -I empfile3 -T 500
```

Programmatic Interfaces

Exit Status Code Handling

`import` sets the exit status code to 0, only if it successfully imports all the specified rows from the input file into the target table. (Complete success).

Note. In Release 2.0, under these conditions, `import` wrote a message to `Stdout` displaying "Import Completed Successfully". For Release 2.X, under these conditions, `import` will write the same message to `Stdout`.

`import` sets the exit status code to 10, if it attempts to import all specified rows from the input file, encounters one or more ignorable errors (parsing or execution errors), does not encounter any non-ignorable errors, and does not exceed any error threshold. (A qualified success).

Note. In Release 2.0, when `import` reached the end of all specified rows and encountered only ignorable errors, Import wrote a message to `Stdout` displaying "Import Completed Successfully". For Release 2.X, under these conditions, `import` will instead write a message displaying "Import Completed with some non-fatal errors"

`import` sets the exit status code to 20, if one or more rows are successfully imported, but it encounters a non-ignorable error or exceeds an error threshold. (Some success, but a serious problem encountered).

Note. In Release 2.0, under these conditions, `import` wrote a message to `Stdout` displaying how many rows were imported, but did not write a message displaying "Import Completed …" In Release 2.X, `import` will do the same.

`import` sets the exit status code to 30 if the input file is readable, but is a zero-length file. (No success and no ignorable errors).

Note. In Release 2.0, under these conditions, Import wrote both a message saying "Rows Imported = 0" and a message saying "Import Completed Successfully" to Stdout . For Release 2.X, under these conditions, Import will write only the message "Rows Imported = 0".

`import` sets the exit status code to 40, if it attempts to import all specified rows from the input file, encounters one or more ignorable errors, does not encounter any non-ignorable errors, and no rows are successfully imported. (No success and ignorable errors encountered).

Note. In Release 2.0, under these conditions, `import` wrote a message to `Stdout` displaying "Import Completed Successfully". For Release 2.X, under these same conditions, `import` will instead write a message displaying "Import Completed with some non-fatal errors".

`import` sets the exit status code to 50, if no rows are successfully imported and it encounters a non-ignorable error or exceeds an error threshold. (No success and a serious problem encountered).

Note. In Release 2.0, under these conditions, `import` wrote a message to `Stdout` displaying how many rows were imported, but did not write a message displaying "Import Completed". In Release 2.X, `import` will do the same.

`import` sets the exit status code to 70, if it detects any errors opening, reading, or writing of input or output files, or other serious problems that prevent it from starting to import the first row from the input file. (Serious problems that prevent the import from starting)

Note. In Release 2.0, under these conditions, `import` will write the same messages to `Stdout` as was done by Release 2.0.

`import` sets the exit status code to 90, if it detects any invalid option or combination of options..

Note. In Release 2.0, under these conditions, `import` will write the same messages to `Stdout` as was done by Release 2.0.

To summarize these exit status codes:

0	Complete success
10	A qualified success
20	Some success, but a serious problem encountered

30	No success and no ignorable errors
40	No success and ignorable errors encountered
50	No success and a serious problem encountered
70	Serious problems that prevent the import from starting
90	Invalid option or combination of options

File permissions

When `import` creates any of the error output files, `import` ensures that the permissions are set so that read and write permission is granted only for the current user and his/her group because these files may have user data that is confidential. Read and write permission for the current user and/or the group may be further restricted depending on the umask setting at the time of the file creations. If any of the error output files is created under the /G directory, the Guardian file access codes shall be set to "CCCU", which will prevent any user other than the owner and his/her group from reading or writing to the file.

Displaying messages

`import` uses the existing error message to display capability when it encounters any error.

Output File Consideration

Non-ignorable execution errors

Inserting a row into a table may be a multi-step process. This involves inserting the row into the base table and may also require adding a row to one or more index tables, checking referential integrity constraints, performing trigger actions,and so on. Some execution errors are detected after a row has been inserted into the target's base table. These include any uniqueness constraints that are enforced by an index, referential integrity constraints, and errors detected while performing trigger actions. Other such errors include those returned by DP2, TMF, and other low level components such as file full conditions, audit file full, or network errors. If any error is encountered after the row has been inserted into the base table, the error cannot be ignored and the IMPORT utility shall terminate just as it does in the current release.

The row contents of the error log file are the same as the input file. However, the position of rows may differ.

Rows in the parsing error log file look exactly like the corresponding row in the input file. However, rows in the execution error log file may not look like the corresponding row in the input file.

Four notable cases are:

1. The format of numbers may be changed.
   For example: 123456.7 in the original input file may appear as 1.234567E+05 in the error log file.
2. The value of some floating point numbers may not be exactly the same (though extremely close). This is due to an internal conversion to binary format and then back to floating point ASCII format for logging to the error log file.
3. Any string which includes the field delimiter, the row delimiter, or the field-qualifier character may appear different.
   For example, if the field qualifier is a double-quote character (")

   Use a double-quote("") *<in the original input file>*

   will reflect as

   "Use a double-quote(")" *<in the error log file>*
4. If the format file is said to skip column 1 of each row in the original input file, the data in column 1 will not be displayed in the execution error log file.
   For example:
   "Data for a manager", 378456, Comm. Dept, 125 *<in the original input file>*

   will reflect as

   , 378456, Comm. Dept, 125 *<in the error log file>*

Note. The leading comma (Field Delimiter character) indicates that the first field is missing.

If the format file is said to skip a column in the middle of each row, the same would be indicated by consecutive Field Delimiter characters in execution error log rows.

If the input file has fixed-width columns instead of field delimiters, the rows in the execution error log file would contain all NullValue characters (as specified in the format file or space by default) for any skipped columns.

Although the appearance of such rows will be different in the execution error log than in the input file, these rows in the error log can still be used to import to the destination table without requiring you to fix these appearance differences. The real problem that caused the row to get an execution error would need to be fixed by you. However, these appearance differences do not require fixing before the row can be imported. Such appearance differences do not affect the column values in the destination table. The only possible exception to this would be if the miniscule change in the value of a floating point number is considered to be significant.

Examples of import

- Example 1 shows an import of data from a delimited file. This example shows the schema of the table to be loaded, the data to be loaded (the input data file), the format file describing the data, and the `import` command used to load it.

  Table Schema

  This statement creates the target table, COMPANY:

```
CREATE TABLE company
    ( id                INT NOT NULL
      ,company          VARCHAR (176)
      ,phone            VARCHAR (12)
      ,fax              VARCHAR (12)
      ,PRIMARY KEY      (id)
    );
```

  Data to Input

  The input file, COINPUT, contains records like this:

```
1,"Test String 1","111-222-3333","222-333-4444"
2,"Test String 2","111-222-3333","222-333-4444"
3,"Test String 3","111-222-3333","222-333-4444"
4,"Test String 4","111-222-3333","222-333-4444"
5,"Test String 5","111-222-3333","222-333-4444"
6,"Test String 6","111-222-3333","222-333-4444"
7,"Test String 7","111-222-3333","222-333-4444"
8,"Test String 8","111-222-3333","222-333-4444"
```

  Format File Describing the Data

  Create a format file, FORMFILE, which consists of [DATE FORMAT], [COLUMN FORMAT], and [DELIMITED FORMAT] sections:

```
[DATE FORMAT]
DateOrder=MDY
DateDelimiter=/
TimeDelimiter=:
FourDigitYear=Y
DecimalSymbol=.

[COLUMN FORMAT]
col=id,N
col=company,N
col=phone,N
col=fax,N

[DELIMITED FORMAT]
FieldDelimiter=,
RowDelimiter=\n
Qualifier="
```

  import Load Command

This `import` command imports data into the COMPANY table from the delimited input file named COINPUT using the format file FORMFILE:

```
import cat.sch.company -I coinput -U formfile -W DELIM
```

- Example 2 shows an import into the same COMPANY table from a fixed width file. This example shows the data to be loaded (the input data file), the format file describing the data, and the `import` command used to load it.

  Data to Input

  The input file, COINPUT_FX, contains records like this:

```
00000000001,"Test String 3456","111-222-3333","444-555-6666"
00000000002,"ibm             ","408-111-2222","408-222-3333"
00000000003,"apple           ","408-222-1111","408-333-2222"
00000000004,"tandem          ","408-285-5000","408-285-2227"
00000000005,"diyatech        ","510-111-2222","510-222-3333"
```

  Format File Describing the Data

```
[DATE FORMAT]
DateOrder=MDY
DateDelimiter=/
TimeDelimiter=:
FourDigitYear=Y
DecimalSymbol=.

[COLUMN FORMAT]
col=id,N
col=company,N
col=phone,N
col=fax,N

[FIXED WIDTH FORMAT]
col=id,1,11
col=company,14,16
col=phone,33,12
col=fax,48,12
RecordLength=61
```

  All commas and double-quote characters in the input file are ignored because they are not covered by any of the col= entries in the [FIXED WIDTH FORMAT] section of the format file. Also, in this example the RecordLength value of 61 includes one newline character at the end of each input record.

  import Load Command

  This `import` command imports data into the COMPANY table from the fixed width input file named COINPUT_FX using the format file FORMFILE:

```
import cat.sch.COMPANY -i COINPUT_FX -u FORMFILE -w FIXED
```

- Example 3 shows an import into TABLE_2 from a fixed width file with data that is not separated by double-quote characters. This example shows the schema of the table to be loaded, the data to be loaded (the input data file), the format file describing the data, and the `import` command used to load it.

Table Schema

This statement creates the TABLE_2 table:

```
CREATE TABLE table_2
    (COL1   CHAR(5)
    );
```

Data to Input

The input file, COINPUT_FX2, contains records like this:

```
0123456789012345
ABCDEFGHIJKLMNPQ
0123456789012345
```

Three hidden spaces are at the end of each line.

Format File Describing the Data

```
[DATE FORMAT]
NormalizeDate = y

[COLUMN FORMAT]
col=col1

[FIXED WIDTH FORMAT]
FileIsBinary=N
RecordLength =5
col=col1,1,5
```

import Load Command

This `import` command imports data into the table_2 table from the delimited input file named COINPUT using the format file FORMFILE:

```
import cat.sch.table_2 -I coinput_fx2 -U formfile -W FIXED
```

- Example 4 illustrates how to use new options and analyze the output files.

Table Schema

This statement creates the target table, cat.sch.Xample:

```
CREATE TABLE cat.sch.Xample
        (  C1 INT          NOT NULL PRIMARY KEY
           , C2 CHAR(8)    NOT NULL
           , CONSTRAINT cnd1 CHECK (c1 > 5)
         );
```

Data to Input

Suppose you have an input file named X.in , containing the following 10 lines

```
11,12345678
A8,22345678
11,82345678
14,423456789
17,52345678,9
15,62345678
27,72345678
5,32345678
BZ,XYZ
55,12345678
```

```
Execute the following command:
 import cat.sch.Xample -i X.in -L 50  -E X.perrs -PM
X.perrmsg \
-XL 5 -XE X.xerrs -XM X.xerrmsg \
-SF X.sum -SI 10
```

Output

```
NonStop SQL/MX Import Utility 2.3
(c) Copyright 2007 Hewlett-Packard Development Company, LP.
Rows Imported = 4
Import Completed with some non-fatal errors
```

A select statement on the table would show the following:

```
select * from cat.sch.Xample;
C1           C2
-------- --------
 11      12345678
 15      62345678
 27      72345678
 55      12345678
--- 4 row(s) selected.
```

The X.perrs file would contain:

```
A8,22345678
14,423456789
17,52345678,9
BZ,XYZ
```

The X.perrmsg file would contain:

```
*** ERROR[20081] Row number 2 and column number 1 could not
be processed.  Column Data: A8

*** ERROR[20291] The data specified in row number 4 column
number 2 is longer than the actual column size definition.
Column Data:123456789.

*** ERROR[20070] Columns in the datafile are not correct.
Columns found so far: 3

*** ERROR[20081] Row number 9 and column number 1 could not
be processed.  Column Data: BZ
```

The X.xerrs would contain:

```
11,82345678
5,32345678
```

The X.xerrmsg file would contain:

```
*** ERROR[8102] The operation is prevented by a unique
constraint.

*** ERROR[8101] The operation is prevented by check
constraint CAT.SCH.CND1 on table CAT.SCH.XAMPLE.
```

The results summary file, X.sum, would contain:

```
Import Results Summary
Import Process ID: 4060
Input File or Process Name: X.in
Start time: Mon Oct 15 13:12:43 2007
Rows to be skipped initially: 0

Rows read so far = 10
Rows skipped = 0
Elapsed time = 0:0:0.088011
Elapsed time since skipping initial rows = 0:0:0.088011
Rows Imported Successfully = 4
Rows ignored due to parsing errors = 4
Rows ignored due to execution errors = 2
Parsing Error Log File Name: X.perrs
Parsing Error Messages File Name: X.perrmsg
Execution Error Log File Name: X.xerrs
Execution Error Messages File Name: X.xerrmsg
Import Completed with some non-fatal errors
```

- Example 5 illustrate –LES options, which takes default value of 1000 each for –L and –XL and auto generates the output error file and error message file and prints the summary to the stdout at the end.

```
import cat.sch.Xample -I x.in -LES
NonStop SQL/MX Import Utility 2.3
(c) Copyright 2007 Hewlett-Packard Development Company, LP.

Rows Imported = 4

Import Results Summary
Import Process ID: 5380
Input File or Process Name: x.in
Start time: Thu Oct 18 15:28:30 2007
Rows to be skipped initially: 0

Rows read so far = 10
Rows skipped = 0
Elapsed time = 0:0:0.187612
Elapsed time since skipping initial rows = 0:0:0.187612
Rows Imported Successfully = 4
Rows ignored due to parsing errors = 5
Rows ignored due to execution errors = 1
Parsing Error Log File Name: PE5380
Parsing Error Messages File Name: PE5380
Execution Error Log File Name: XE5380
Execution Error Messages File Name: XE5380
Import Completed with some non-fatal errors
```

Support for restarting import

To support the ability to restart `import` after it terminates due to excessive parsing errors, excessive execution errors, or non-ignorable errors, the following information is appended to the results summary file before `import` terminates:

- The number of rows with parsing errors.
- The number of rows with execution errors.

- The number of rows imported successfully.
- A line saying "Import Completed Successfully" or "Import Completed with non-fatal errors" if `import` has reached the end of the input data without encountering any non-ignorable errors.
- The name for the parsing error log file (if any).
- The name for the parsing error message log file (if any).
- The name for the execution error log file (if any).
- The name for the execution error message log file (if any).

If `import` terminates due to any non-ignorable error, the following information is also appended:

- The value, X, to specify with the `-F` <first-row> option on a subsequent execution of `import` to have the subsequent `import` pick up where the first `import` left.
- A warning message if the last Y number of rows in the parsing error output file were detected on or after row X (see the previous bulleted point) in the input file (if Y is greater than 0).
- A warning message if the last Z number of rows in the execution error output file were detected on or after row X (see previous bullet) in the input file (if Z is greater than 0).

When transactions are being used, X is the number of rows read by `import` as of the last successful transaction.

When transactions are not being used (while using the Fast Loading technique -- known as Side-Tree inserts by project personnel) and termination is due to exceeding a user-specified error threshold, X is the number of rows read by `import` before the threshold was exceeded.

When transactions are not being used and termination is due to a non-ignorable error other than exceeding a user-specified error threshold, X is zero.

The two warning messages in the above list are needed for you to understand that if you fix the problem in the error output file(s) and import those rows before restarting `import` with the original input file using the `-F` <first-row> option, the restarted `import` will attempt to import those rows for the second time (unless you delete those rows from the input file).

Note. If the associated error messages are interspersed in the same file as the rows in error, the values reported for Y and Z will count each associated error message as if it were a row (even if the input file is of type FIXED and the associated error messages are variable in length.)

The rows with errors reported after the row, indicated for <first-row> must be dealt with carefully. If you fix such rows in the error log file(s), import the fixed rows from the error log file(s), but do not delete the rows in the original input file before restarting import with the original input file (using the -F <first-row> option). The parsing errors and/or execution errors will still be detected for those rows. The easiest way to handle such rows would be to delete them from the error log before using it as an input file and fix those rows in the original input file before restarting import with the original input file (using the -F option). If the original input file cannot be easily modified, it is recommended that you delete such rows from the error log and let those errors occur again when import is restarted with the original input file. Presumably, those rows would be detected before the new error thresholds are reached.

A sample results summary file is as follows:

Note. The lines that can be repeated are shown in bold in the following sample summary file.

```
Input File or Process Name: myinput_for_MYTABLE
Start time: Mon May  2 15:33:13 CDT 2005
Rows to be skipped initially: 201986
Rows read so far = 2579352
Rows skipped = 359827
Elapsed time = 01:46:37.18
Elapsed time since skipping initial rows = 01:13:54:30
Rows Imported Successfully = 2570000
Rows ignored due to parsing errors = 4
Rows ignored due to execution errors = 2
Parsing Error Log File Name: MYTABLE_perrs
Parsing Error Messages File Name: MYTABLE_perrs
Execution Error Log File Name: MYTABLE_xerrs
Execution Error Messages File Name: MYTABLE_xerrs
You should specify -F 2570003 if you want to restart IMPORT
and have it start where this import operation left off.
```

▲ **WARNING.** The last two rows of the error output file, MYTABLE_perrs, are found on or after row number 2570003 in the input file.

▲ **WARNING.** The last 1 rows of the error output file, MYTABLE_xerrs, are found on or after row number 2570003 in the input file.

Note. A line displaying one of the following will be appended to the end of the results summary file whenever `import` reaches the end of the input data without encountering any non-ignorable error conditions. For this sample results file, neither message would be appended as this example is for a case where `import` had to roll back to the last transaction, and must have encountered some non-ignorable error condition.

```
Import Completed with some non-fatal errors
Import Completed Successfully
```

INFO Operation

Considerations for INFO
Examples of INFO

INFO is an OSS command-line utility run from `mxtool` that displays information about SQL/MX files. INFO displays the Guardian file name, the ANSI name, the ANSI namespace, and the object schema version.

```
INFO [\node.]\$volume.subvol.filename
```

`[`*`node`*`.]\$`*`volume.subvol.filename`*

is the Guardian qualified file set list that specifies the set of Guardian files that is being queried. It must be fully qualified with the volume and subvolume names.

In this four-part name, *`node`* is the name of a node of a NonStop server, *`$volume`* is the name of a disk volume, *`subvol`* is the name of a subvolume that begins with the letters ZSD, and *`filename`* is the automatically generated name of a Guardian table that ends with “00” or “01” (zero zero or zero one).

You can use the “*” Guardian wild card in the volume, subvolume, and file name.

Considerations for INFO

Security Considerations

To perform INFO on files on other nodes, the remote system must be available.

If INFO tries to access objects that have a schema version that is greater than the NonStop SQL/MX software version (MXV) of the local node, you receive a versioning error.

Other Considerations

Use INFO to request specific SQL/MX information for a Guardian file name without writing queries against metadata. Use INFO to obtain this information if metadata is unavailable.

You can also use the SHOWLABEL command to get more information on a Guardian file. See SHOWLABEL Command on page 4-99 for more information.

This information is displayed:

- The ANSI name
- The ANSI namespace, including table namespace (which includes views and stored procedures), index namespace, and trigger namespace
- The object schema version

Examples of INFO

- These are examples of INFO queries:

```
mxtool INFO \\figaro.\$data*.*svol*.*
mxtool INFO \\figaro.\$*.*.*
mxtool INFO \\figaro.\$vol.subvol.file*
mxtool INFO \\figaro.\$vol.subvol*.file*00
```

- This is an example of an INFO query on SQL/MX catalogs using a wild card:

```
mxtool INFO \$DATA09.ZSD0.CATSYS0*
File Name: \FIGARO.$DATA09.ZSD0.CATSYS00
Object Schema Version: 1200
Ansi Name: NONSTOP_SQLMX_FIGARO.SYSTEM_SCHEMA.CATSYS
Ansi NameSpace: TA

File Name: \FIGARO.$DATA09.ZSD0.CATSYS01
```

- This is an example of an INFO query on a volume that does not exist:

```
mxtool INFO \$VOL.SUBVOL.FILE
*** ERROR[20362] Error 14 was returned while validating the
file set list specified by "$VOL.SUBVOL.FILE".
```

- This is an example of an INFO query on a node that does not exist:

```
mxtool INFO \\SQUAW.\$DATA09.ZSDT6TG2.HSNZ4500
*** ERROR[20373] The specified system does not exist in the
network.
```

- This is an example of an INFO query on a file that is not an SQL/MX object:

```
mxtool INFO \$DATA09.XYZ.MXCMP
*** ERROR[20357] File \FIGARO.$DATA09.XYZ.MXCMP is not an
SQL/MX object.
```

- This is an example of an INFO query on a file that does not exist:

```
mxtool INFO \$DATA09.ZSDT6TG2.HSNZ4500
*** ERROR[20355] File $DATA09.ZSDT6TG2.HSNZ4500 was not
found.
```

mxexportddl Utility

[Considerations for mxexportddl](#)

[Examples of mxexportddl](#)

`mxexportddl` is an OSS command-line utility that captures the metadata and statistics of SQL/MX objects and saves them in the XML format. This utility supports OSS large files (files greater than 2 GB) as output files.

It is used to export the:

- SQL/MX object metadata to an XML file for DDL replication
- SQL/MX table statistics to an XML file for statistics replication

---

Note. The `mxexportddl` utility is enhanced and is available only on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

The XML files generated by the older version of the `mxexportddl` utility cannot be imported with the newer version of the `mximportddl` utility and vice versa.

Exporting Metadata and Statistics of SQL/MX Objects

A catalog is a logical object; it is a collection of schemas. A schema is a logical objects that has a collection of database objects such as tables, indexes, views, and stored procedures.

- Exporting a catalog includes exporting all the subordinate objects in its hierarchy such as schemas, tables, and the subordinate objects of the tables.
- To export objects on a remote node, the catalog must be registered using the REGISTER CATALOG command. A catalog is not visible to a remote node until it is registered.
- Exporting a schema includes exporting all its subordinate tables, views, and stored procedures.
- Exporting a table includes:
  - Non-droppable constraints
  - Droppable constraints unless the CONSTRAINTS OFF option is enabled
  - All table partitions excluding data
  - All indexes and index partitions excluding data
  - All triggers and referential integrity constraints
  - Statistics of the table unless the STATS OFF option is enabled
- Indexes cannot be explicitly specified in the `mxexportddl` utility. They are subordinate to the table object and can only be exported with the parent table.

- A constraint cannot be specified explicitly in the `mxexportddl` utility. To export a constraint, the table including the constraint must be exported. The constraints that are exported vary depending on whether the CONSTRAINTS option is set to ON or OFF.
- Views can be implicitly exported with the entire schema and not only by the table.
- System defaults are always exported.

```
mxexportddl
-HELP
|
{
sqlmx-object-spec-list
-XMLFILE xml-file-name
[-CONSTRAINTS ON | OFF]
[-LOGFILE log-file-name]
[-STATS ON | OFF ]
[-CLEAR ON | OFF ]
}
```

`mxexportddl`

must be lowercase.

`-HELP`

directs `mxexportddl` to display the syntax.

`sqlmx-object-spec-list`

is the list of SQL/MX object names with their corresponding object types. The *`sqlmx-object-spec-list`* can be specified as:

```
( sqlmx-object-type sqlmx-object-name-list
[ sqlmx-object-type sqlmx-object-name-list] ... )
```

The *`sqlmx-object-type`* is one of these object types:

```
{ -CAT | -CATALOG }
{ -SCH | -SCHEMA }
{ -TAB | -TABLE }
```

`-CAT | -CATALOG`

is the object type for a catalog.

`-SCH | -SCHEMA`

is the object type for a schema.

`-TAB | -TABLE`

is the object type for a table.

The *`sqlmx-object-name-list`* is:

`(` *`sqlmx-object-name`* `[`*`sqlmx-object-name`*`] ... )`

`sqlmx-object-name` is a fully-qualified SQL/MX object name of the specified *`sqlmx-object-type`*. The specified object and all subordinate objects are exported. Wild cards are not permitted in the *`sqlmx-object-name`*. For delimited names, use ' \" ' to represent a quote. If a schema is "CATALOG"."SCHEMA", you must represent the schema as \"CATALOG\".\"SCHEMA\". If you do not use ' \" ', the osh shell strips the required ' " ' characters.

`-XMLFILE` *`xml-file-name`*

specifies the name of the XML file, which is generated by the `mxexportddl` utility. The *`xml-file-name`* must be a valid OSS file name.

`-CONSTRAINTS  ON | OFF`

specifies whether droppable constraints need to be exported or not. The default is ON.

`ON`

exports the droppable and non-droppable constraints of the table.

`OFF`

exports only the non-droppable constraints of the table.

`-LOGFILE` *`log-file-name`*

redirects the screen logs of the `mxexportddl` utility into an OSS file. The *`log-file-name`* must be a valid OSS file name.

`-STATS ON |OFF`

specifies if the statistics of the tables needs to be exported. The default is ON.

`ON`

exports the statistics of the tables.

`OFF`

does not export the statistics of the tables. Only the metadata of the SQL/MX objects is exported.

`-CLEAR ON |OFF`

specifies if the `mxexportddl` utility can overwrite the specified XML file if it already exists. The default is OFF.

`ON`

overwrites the specified XML file if it already exists.

`OFF`

does not overwrite the specified XML file. If the file already exists, the write operation fails.

Considerations for mxexportddl

You can edit the XML file manually using simple text editors. However, this method can be error-prone and is not recommended.

Because the schema owner information in the XML file can be updated, other users can import the schema and its objects.

Remote catalogs must be registered manually before executing `mxexportddl`.

`mxexportddl` supports the RI actions CASCADE/SET NULL/SET DEFAULT in addition to NO ACTION and RESTRICT.

Supported by mxexportddl

- System defaults
- Tables and associated objects such as indexes and partitions
- Constraints: check, not null, primary key, and unique
- Table statistics that includes physical statistics (index level, nonempty block count, and EOF), histograms, and histogram intervals
- Referential integrity constraints, views, triggers, and stored procedures
- For a table containing the IDENTITY column, the internal Sequence Generator attributes will also be stored in the XML file

Not Supported by mxexportddl

- SQL/MP tables and aliases

Examples of mxexportddl

- To export the catalog `cat1` and schema `cat2.sch1` and `cat2.sch2`:

```
mxexportddl -cat cat1 -sch cat2.sch1 cat2.sch2 -xmlfile
export.xml
```

- To export the catalog `cat1` without statistics:

```
mxexportddl -cat cat1 -xmlfile export.xml -stats off
```

MXGNAMES Utility

[Considerations for MXGNAMES](#)
[Examples of MXGNAMES](#)

MXGNAMES is a Guardian program that is run from a TACL prompt or an OBEY command file.

It converts one or more ANSI table names into a list of corresponding Guardian file names, appropriately formatted for TMF or BACKUP/RESTORE 2.

For table containing an IDENTITY column, the SG Table location is also displayed in the MXGNAMES output.

```
MXGNAMES -HELP  |

input [-output=output-file-name] output-format
      [-node=node-name] [-length=file-length] [-nocomment]

input is
    -SQLnames=SQL-table-name-list-file
  | -Showddl=SHOWDDL-file-name
  | SQLMX-table-name

output-format is
  -BR2
  -TMF
```

output-file-name

is the name of a nonexistent Guardian disk file to which the output should be written. If you do not specify an output file, output is written to the screen. If the file already exists, or cannot be created, an error is returned.

SQL-table-name-list-file

is an EDIT format file that consists of a list of a list of fully qualified ANSI table names, one per line. MXGNAMES ignores blank lines.

Because of the 255 character limit for EDIT files, MXGNAMES cannot support table names in the file whose overall length is greater than 255 characters, including the dots separating the catalog, schema, and table name portions. You must specify such names individually on the MXGNAMES command line.

SHOWDDL-file-name

is an EDIT file to be used as input that contains SHOWDDL output for one SQLMX table.

SHOWDDL output is normally saved to an OSS format text file. You must use CTOEDIT to convert this file to an EDIT file before you can use it as input to MXGNAMES. You cannot convert the SHOWDDL file to an EDIT file if it contains an ANSI table name whose overall length is greater than 255 characters, including the dots separating the catalog, schema, and table name portions. You must specify such names individually on the MXGNAMES command line.

SQLMX-table-name

is a single fully qualified SQLMX table name entered directly on the command line

output-format

indicates what subsystem the output file is to be used with:

-BR2

indicates output should be formatted for use with the BR2 RESTORE command.

The resulting LOCATION clauses contain both the complete source and target file names, where the target *node-name* specified in the -node option is substituted in the target location, and the target volume, subvolume and file name match the source exactly.

-TMF

indicates output should be formatted for use with TMF.

TMF has limits on the size of a command file. The maximum size of a command file is 28,000 bytes. You can use the -length option to control the size of the output file, while allowing for other text to be manually added to the command file without exceeding the 28,000 byte limit.

Resource forks must be explicitly dumped and recovered. Therefore, the output contains file names listed as wild cards that include the resource forks.

node-name

indicates what node name, including the backslash, should be appended to the target location for the RESTORE command. If *output-format* is -BR2, this argument is required. If *output-format* is -TMF, this option is ignored.

file-length

is an integer representing the maximum size of the output file in bytes. If the total amount of data exceeds this amount, MXGNAMES generates additional files by appending numbers to the output file name, truncating the output file name, if necessary. If you do not specify the -length option, all output is placed in a single file. The value you specify for *file-length* must be at least 1000.

-nocomment

indicates that no comments are to be included in the output file. TMF commands allow comments in the text, which is the default if the *output-format* is -TMF. If the output format is `-BR2`, no comments are included, regardless of whether you specify `-nocomment`.

`-HELP`

displays help for the utility.

Considerations for MXGNAMES

You must have READ/WRITE access to the Guardian subvolume where you are executing MXGNAMES.

Input and output of MXGNAMES is in standard Guardian EDIT files, which have these characteristics:

- Guardian file code of 101
- Maximum line length of 255 characters

You can use other tools or programs to capture data for use with MXGNAMES. To use the captured data with MXGNAMES, you might need to convert the file to the EDIT format. You can use tools such as CTOEDIT to perform the conversions.

Temporary Work Files

MXGNAMES might generate temporary workfiles during execution. These files are placed in the current volume and subvolume. You must have authority to read and write to this location, or MXGNAMES generates an error and halts execution.

Examples of MXGNAMES

- Suppose that these SQLMX tables and indexes exist on the system:

```
create table CAT.SCH.T126A
   ( c1 INT not null
   , c2 TIMESTAMP default current_timestamp not null
   , c3 CHAR(4) default 'abcd'
   , c4 SMALLINT not null
   , PRIMARY KEY (c1,c2) )
location $VOL1.ZSDA126A.BXNL1R00
partition
   ( add first key (1r00) location $VOL2.ZSDA126A.BXNL2R00
   , add first key (2r00) location $VOL3.ZSDA126A.BXNL3R00
   , add first key (3r00) location $VOL4.ZSDA126A.BXNL4R00
   , add first key (4r00) location $VOL5.ZSDA126A.BXNL5R00
   , add first key (5r00) location $VOL6.ZSDA126A.BXNL6R00 )
store by primary key;

create index T126A_NDX1 on CAT.SCH.T126A(c4)
location $vol1.ZSDA126a.qdxwg100
partition
   ( add first key (100) location $VOL2.ZSDA126A.QDXWG200
   , add first key (500) location $VOL3.ZSDA126A.QDXWG300
   , add first key (700) location $VOL4.ZSDA126A.QDXWG400 )
;

create table CAT.SCH.T126B
   ( c1 timestamp default current_timestamp not null
```

```
   , c2 INT not null
   , c3 VARCHAR (30)
   , c4 SMALLINT not null
   , PRIMARY KEY (c4, c1) )
location $VOL1.ZSDA126B.BXNW1R00
hash partition by (c4)
   ( add location $VOL2.ZSDA126B.BXNW2R00
   , add location $VOL3.ZSDA126B.BXNW3R00
   , add location $VOL4.ZSDA126B.BXNW4R00 )
;
create unique index T126B_NDX1 on CAT.SCH.T126B(C2, C1)
LOCATION $vol1.ZSDA126b.qdx1g100
hash partition by (c2)
   ( add location $VOL2.ZSDA126B.QDX1G200
   , add location $VOL3.ZSDA126B.QDX1G300
   , add location $VOL4.ZSDA126B.QDX1G400
   , add location $VOL5.ZSDA126B.QDX1G500 )
;
```

- This example prepares a list of file names to be used with TMF. The input is a list of fully qualified SQL names.

  ```
  MXGNAMES -SQLNames=$VOL1.SQLSTUFF.SQLNAMES -output=NAMELIST -TMF
  ```

  Suppose the contents of the file SQLNAMES are:

  ```
  CAT.SCH.T126A
  CAT.SCH.T126B
  ```

  The output of the command is a file called NAMELIST which contains:

  ```
  (-- Table CAT.SCH.T126A --&
  $VOL1.ZSDA126A.BXNL1R*,&
  $VOL2.ZSDA126A.BXNL2R*,&
  $VOL3.ZSDA126A.BXNL3R*,&
  $VOL4.ZSDA126A.BXNL4R*,&
  $VOL5.ZSDA126A.BXNL5R*,&
  $VOL6.ZSDA126A.BXNL6R*,&
  -- Index T126A_NDX1 on CAT.SCH.T126A--&
  $VOL1.ZSDA126A.QDXWG1*,&
  $VOL2.ZSDA126A.QDXWG2*,&
  $VOL3.ZSDA126A.QDXWG3*,&
  $VOL4.ZSDA126A.QDXWG4*,&
  -- End of table CAT.SCH.T126A--&
  -- Table CAT.SCH.T126B--&
  $VOL1.ZSDA126A.BXNW1R*,  &
  $VOL2.ZSDA126A.BXNW2R*,  &
  $VOL3.ZSDA126A.BXNW3R*,  &
  $VOL4.ZSDA126A.BXNW4R*,  &
  -- Index T126B_NDX1 on CAT.SCH.T126B-- &
  $VOL1.ZSDA126A.QDX1G1*,&
  $VOL2.ZSDA126A.QDX1G2*,&
  $VOL3.ZSDA126A.QDX1G3*,&
  $VOL4.ZSDA126A.QDX1G4*,&
  $VOL5.ZSDA126A.QDX1G5* &
  -- End of table CAT.SCH.T126B--)
  ```

- This example prepares a list of file names to be used with TMF. The input is SHOWDDL text.

  ```
  MXGNAMES -Showddl=$VOL1.SQLSTUFF.SHOWD123 -output=NAMELST -TMF
  ```

  Suppose the contents of the file SHOWD123 are:

  ```
  CREATE TABLE CAT.SCH.T126A
    (
  ```

```
    C1  INT NO DEFAULT -- NOT NULL NOT DROPPABLE
  , C2  TIMESTAMP(6) DEFAULT CURRENT_TIMESTAMP   -- NOT NULL NOT
DROPPABLE
  , C3  CHAR(4) CHARACTER SET ISO88591 COLLATE   DEFAULT DEFAULT
_ISO88591'abcd'
  , C4  SMALLINT NO DEFAULT                      -- NOT NULL NOT
DROPPABLE
  , CONSTRAINT CAT.SCH.T126A_106009919_0001 PRIMARY KEY (C1 ASC, C2 ASC)
    NOT DROPPABLE
  , CONSTRAINT CAT.SCH.T126A_106009919_0000 CHECK (CAT.SCH.T126A.C1 IS
NOT NULL
      AND CAT.SCH.T126A.C2 IS NOT NULL AND CAT.SCH.T126A.C4 IS NOT NULL)
NOT
      DROPPABLE
  )
  LOCATION \NSK.$VOL1.ZSDA126A.BXNL1R00
  NAME PART_A_Z_1
  PARTITION
  (
    ADD FIRST KEY (1000)
      LOCATION \NSK.$VOL2.ZSDA126A.BXNL2R00
      NAME PART_A_B_C
  , ADD FIRST KEY (2000)
      LOCATION \NSK.$VOL3.ZSDA126A.BXNL3R00
      NAME PART_D_E_F
  , ADD FIRST KEY (3000)
      LOCATION \NSK.$VOL4.ZSDA126A.BXNL4R00
      NAME PART_J_K_L
  , ADD FIRST KEY (4000)
      LOCATION \NSK.$VOL5.ZSDA126A.BXNL5R00
      NAME PART_M_N_O
  , ADD FIRST KEY (5000)
      LOCATION \NSK.$VOL6.ZSDA126A.BXNL6R00
      NAME PART_P_Q_R
  )
  STORE BY (C1 ASC, C2 ASC)
  ;
CREATE INDEX T126A_NDX1 ON CAT.SCH.T126A
  (
    C4 ASC
  )
  LOCATION \NSK.$VOL1.ZSDA126A.QDXWG100
  NAME PART_V_W_X  PARTITION
  (
    ADD FIRST KEY (100)
      LOCATION \NSK.$VOL2.ZSDA126A.QDXWG200
      NAME PART_S_T_U
  , ADD FIRST KEY (500)
      LOCATION \NSK.$VOL3.ZSDA126A.QDXWG300
      NAME PART_Y_Z_1
  , ADD FIRST KEY (700)
      LOCATION \NSK.$VOL4.ZSDA126A.QDXWG400
      NAME PART_A_Z_1
  )
  ;
```

The resulting contents of file NAMELST would be:

```
(-- Table CAT.SCH.T126A  -- &
$VOL1.ZSDA126A.BXNL1R*,&
$VOL2.ZSDA126A.BXNL2R*,&
$VOL3.ZSDA126A.BXNL3R*,&
$VOL4.ZSDA126A.BXNL4R*,&
$VOL5.ZSDA126A.BXNL5R*,&
$VOL6.ZSDA126A.BXNL6R*,&
 &
```

```
-- Index T126A_NDX1 on CAT.SCH.T126A  -- &
$VOL1.ZSDA126A.QDXWG1*,&
$VOL2.ZSDA126A.QDXWG2*,&
$VOL3.ZSDA126A.QDXWG3*,&
$VOL4.ZSDA126A.QDXWG4* &
-- End of table CAT.SCH.T126A ) --
```

- This example prepares a list of file names to be used with TMF. The input is an SQL table name.

  ```
  MXGNAMES CAT.SCH.T126A -output=NAMELSTX -TMF
  ```

  The contents of NAMELSTX are identical to the output file of the second example.

- This example uses MXGNAMES with TMF. The input is a list of SQL names with the file length specified.

  ```
  MXGNAMES -SQLNames=$VOL1.SQLSTUFF.SQLNAMES -output=NAMELIST -TMF
         -length=28000
  ```

  The above use of the `-length` option specifies that output files should be limited to a length of 28000 bytes. If the output exceeds 28000 bytes, the first additional file generated is called NAMELIS2. If ten output files are needed, the tenth output file is called NAMELI10.

  You can use the file length to allow for additional text to be added to the TMF command file, in addition to the text generated by MXGNAMES, without exceeding TMF's 28000 byte limit. The minimum file length allowed is 1000 bytes.

  If MXGNAMES generates multiple output files, a duplicate file name error can occur on one of these files. To avoid duplicates you should:

  - Specify shorter output file names, so that the extra digits can be appended without overwriting characters from the original name.
  - Avoid using digits in the end of the output file name.

  Additional files either do or do not contain comments, depending on whether you use the `-no comment` option.

  Each continuation file will resume with the very next line of output, whether that line is a comment or a file location. Other than the opening parenthesis for each new file, the contents are exactly the same, regardless of the number of files generated.

- This example prepares a list of file names to be used with RESTORE. The input is a list of SQL names.

  ```
  MXGNAMES -SQLNames=$VOL1.SQLSTUFF.SQLNAMES -output=NAMELIST -BR2
         -node=\BNODE
  ```

  Suppose the contents of the file SQLNAMES are as:

  ```
  CAT.SCH.T126A
  CAT.SCH.T126B
  ```

  The output of the command is a file called NAMELIST containing this:

  ```
  LOCATION
  (
  \PNODE.$VOL1.ZSDA126A.BXNL1R00 TO \BNODE.$VOL1.ZSDA126A.BXNL1R00,
  ```

```
\PNODE.$VOL2.ZSDA126A.BXNL2R00 TO \BNODE.$VOL2.ZSDA126A.BXNL2R00,
\PNODE.$VOL3.ZSDA126A.BXNL3R00 TO \BNODE.$VOL3.ZSDA126A.BXNL3R00,
\PNODE.$VOL4.ZSDA126A.BXNL4R00 TO \BNODE.$VOL4.ZSDA126A.BXNL4R00,
\PNODE.$VOL5.ZSDA126A.BXNL5R00 TO \BNODE.$VOL5.ZSDA126A.BXNL5R00,
\PNODE.$VOL6.ZSDA126A.BXNL6R00 TO \BNODE.$VOL6.ZSDA126A.BXNL6R00,
\PNODE.$VOL1.ZSDA126A.QDXWG100 TO \BNODE.$VOL1.ZSDA126A.QDXWG100,
\PNODE.$VOL2.ZSDA126A.QDXWG200 TO \BNODE.$VOL2.ZSDA126A.QDXWG200,
\PNODE.$VOL3.ZSDA126A.QDXWG300 TO \BNODE.$VOL3.ZSDA126A.QDXWG300,
\PNODE.$VOL4.ZSDA126A.QDXWG400 TO \BNODE.$VOL4.ZSDA126A.QDXWG400,
\PNODE.$VOL1.ZSDA126A.BXNW1R00 TO \BNODE.$VOL1.ZSDA126A.BXNW1R00,
\PNODE.$VOL2.ZSDA126A.BXNW2R00 TO \BNODE.$VOL2.ZSDA126A.BXNW2R00,
\PNODE.$VOL3.ZSDA126A.BXNW3R00 TO \BNODE.$VOL3.ZSDA126A.BXNW3R00,
\PNODE.$VOL4.ZSDA126A.BXNW4R00 TO \BNODE.$VOL4.ZSDA126A.BXNW4R00,
\PNODE.$VOL1.ZSDA126A.QDX1G100 TO \BNODE.$VOL1.ZSDA126A.QDX1G100,
\PNODE.$VOL2.ZSDA126A.QDX1G200 TO \BNODE.$VOL2.ZSDA126A.QDX1G200,
\PNODE.$VOL3.ZSDA126A.QDX1G300 TO \BNODE.$VOL3.ZSDA126A.QDX1G300,
\PNODE.$VOL4.ZSDA126A.QDX1G400 TO \BNODE.$VOL4.ZSDA126A.QDX1G400,
\PNODE.$VOL5.ZSDA126A.QDX1G500 TO \BNODE.$VOL5.ZSDA126A.QDX1G500
)
```

- This example prepares a list of file names to be used with RESTORE. The input is SHOWDDL text.

```
MXGNAMES -Showddl=$VOL1.SQLSTUFF.SHOWD123 -output=NAMELST2 -BR2
       -node=\bnode
```

Suppose the contents of the file SHOWD123 are the same as for the second example. The contents of output file NAMELST2 is:

```
LOCATION
(
\PNODE.$VOL1.ZSDA126A.BXNW1R00 TO \BNODE.$VOL1.ZSDA126A.BXNW1R00,
\PNODE.$VOL2.ZSDA126A.BXNW2R00 TO \BNODE.$VOL2.ZSDA126A.BXNW2R00,
\PNODE.$VOL3.ZSDA126A.BXNW3R00 TO \BNODE.$VOL3.ZSDA126A.BXNW3R00,
\PNODE.$VOL4.ZSDA126A.BXNW4R00 TO \BNODE.$VOL4.ZSDA126A.BXNW4R00,
\PNODE.$VOL5.ZSDA126A.BXNW5R00 TO \BNODE.$VOL5.ZSDA126A.BXNW5R00,
\PNODE.$VOL6.ZSDA126A.BXNW6R00 TO \BNODE.$VOL6.ZSDA126A.BXNW6R00,
\PNODE.$VOL1.ZSDA126A.QDX1G100 TO \BNODE.$VOL1.ZSDA126A.QDX1G100,
\PNODE.$VOL2.ZSDA126A.QDX1G200 TO \BNODE.$VOL2.ZSDA126A.QDX1G200,
\PNODE.$VOL3.ZSDA126A.QDX1G300 TO \BNODE.$VOL3.ZSDA126A.QDX1G300,
\PNODE.$VOL4.ZSDA126A.QDX1G400 TO \BNODE.$VOL4.ZSDA126A.QDX1G400
)
```

- This example prepares a list of file names to be used with RESTORE. The input is an SQL table name.

```
MXGNAMES CAT.SCH.T126A -output=NAMELST3 -BR2 -node=\bnode
```

The contents of NAMELST3 are identical to the output file in the sixth example (assuming the definition of table CAT.SCH.T126A is the same).

- In this example, the table contains an IDENTITY column:

```
CREATE TABLE CAT.SCH.TABLE_IDENTITY
  (
    I  LARGEINT GENERATED BY DEFAULT AS IDENTITY
      (START WITH 0 INCREMENT BY 1 MAXVALUE 9223372036854775807
MINVALUE 0 NO CYCLE)
       LOCATION \DMR15.$SYSTEM.ZSDWDPR4.WD4DR900
      -- NOT NULL NOT DROPPABLE
  , CONSTRAINT CAT.SCH.TABLE_IDENTITY_849996266_9538 CHECK
      (CAT.SCH.TABLE_IDENTITY.I IS NOT NULL) NOT DROPPABLE
  )
  LOCATION \DMR15.$SYSTEM.ZSDWDPR4.L26KR900
```

```
  NAME DMR15_SYSTEM_ZSDWDPR4_L26KR900
  ATTRIBUTES BLOCKSIZE 4096
  NO PARTITION
  ;
```

The following command prepares a list of file names to be used with TMF. The input is a SQL table name.

```
$SYSTEM ZMXTOOLS 23> run mxgnames cat.sch.table_identity -TMF
```

mximportddl Utility

[Considerations for mximportddl](#)

[Examples of mximportddl](#)

`mximportddl` is an OSS command-line utility that replicates the DDL definition and statistics of SQL/MX objects.

It is used to import the:

- SQL/MX object metadata from an XML file for DDL replication
- SQL/MX table statistics from an XML file for statistics replication

---

Note. The `mximportddl` utility is available only on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

Importing Metadata and Statistics of SQL/MX Objects

A catalog is a logical object; it is a collection of schemas. A schema is a logical object that has a collection of database objects such as tables, indexes, views, and stored procedures.

- Importing a catalog includes importing all the subordinate objects in its hierarchy such as schemas, tables, and the subordinate objects of the tables.
- Importing a schema includes importing all its subordinate tables, views, and stored procedures.
- Importing a table includes:
  - Non-droppable constraints
  - Droppable constraints unless the CONSTRAINTS OFF option is enabled
  - All table partitions excluding data
  - All indexes and index partitions excluding data
  - All triggers and referential integrity constraints
  - Statistics of the table unless the STATS OFF option is enabled
- Indexes cannot be specified explicitly in the `mximportddl` utility. They are subordinate to the table object and can only be imported with the parent table.
- A constraint cannot be explicitly specified in the `mximportddl` utility. To import a constraint, the table including the constraint must be imported. The constraints that are imported vary depending on whether the CONSTRAINTS option is set to ON or OFF.

- Referential integrity constraints, views, triggers, and stored procedures are referred to as dependency objects. The dependency objects are imported at the end of the `mximportddl` utility to ensure that all parent objects are imported in advance.
- The SHOWDDL ON and SHOWDDLLOC options are provided, the DDL of dependency objects will be written to OSS files. This can be used to manually create the objects later. Dependency objects will not be imported.
- Views can be implicitly imported with the entire schema and not only by the table.
- System defaults are not imported.

```
mximportddl
-HELP
|
{
-XMLFILE xml-file-name
[sqlmx-object-spec-list]
[-MAPFILE map-file-name]
[-CONSTRAINTS ON | OFF]
[-LISTONLY ON | OFF]
[-LOGFILE log-file-name]
[-STATS ON | OFF ]
[-KEEPSTATS ON | OFF]
[-KEEPDDL ON | OFF]
[-KEEPGFN ON | OFF]
[-SHOWDDL ON | OFF]
[-SHOWDDLLOC oss-directory]
}
|
{
-PREPAREMAP
-XMLFILE xml-file-name
-MAPFILE map-file-name
[-LOGFILE log-file-name]
[-CLEAR ON | OFF ]
}
```

`mximportddl`

must be lowercase.

`-HELP`

directs `mximportddl` to display the syntax.

`sqlmx-object-spec-list`

is the list of SQL/MX object names with their corresponding object types. The *`sqlmx-object-spec-list`* is specified as:

```
( sqlmx-object-type sqlmx-object-name-list
[ sqlmx-object-type sqlmx-object-name-list] ... )
```

The *sqlmx-object-type* is one of these object types:

```
{ -CAT | -CATALOG }
{ -SCH | -SCHEMA }
{ -TAB | -TABLE }
```

-CAT | -CATALOG

is the object type for a catalog.

-SCH | -SCHEMA

is the object type for a schema.

-TAB | -TABLE

is the object type for a table.

The *sqlmx-object-name-list* is:

```
( sqlmx-object-name [sqlmx-object-name] ... )
```

sqlmx-object-name is a fully-qualified SQL/MX object name of the specified *sqlmx-object-type*. The specified object and all subordinate objects are imported. Wild cards are not permitted in the *sqlmx-object-name*. For delimited names, use ' \" ' to represent a quote. If a schema is "CATALOG"."SCHEMA", you must represent the schema as \"CATALOG\".\"SCHEMA\". If you do not use the ' \" ', the osh shell strips the required ' " ' characters.

If *sqlmx-object-spec-list* is not provided, all the SQL/MX objects in the XML file will be imported.

-CONSTRAINTS ON | OFF

specifies if droppable constraints need to be imported. The default is ON.

ON

imports the droppable and non-droppable constraints of the table.

OFF

imports only the non-droppable constraints of the table.

-LOGFILE or -LOG *log-file-name*

redirects the screen logs of the `mximportddl` utility into an OSS file. The *log-file-name* must be a valid OSS file name.

`-MAPFILE` or `-MAP` *`map-file-name`*

specifies the name of the MAP file. *`map-file-name`* must be a valid OSS file name.

While importing the XML file, MAP file can be used to map the source catalog or schema to a target catalog or schema. Also, the source node name and volume can be mapped to the target node name and volume. The MAP file template can be generated for the specific XML file, by specifying -PREPAREMAP in the `mximportddl` utility.

A MAP file consists of multiple sections, which are listed below:

`[CATALOG-MAPPING]`

`[SCHEMA-MAPPING]`

`[LOCATION-MAPPING]`

These sections have the following keywords:

```
[CATALOG-MAPPING]
CATALOG <catalog name> = <New catalog name>;
. . .


[SCHEMA-MAPPING]
SCHEMA <catalog name>.<schema name> = <New catalog name>.<new
schema name>;
. . .


[LOCATION-MAPPING]
LOCATION <System name>.<volume name> = <New system name>.<new
volume name>;
. . .
```

`-XMLFILE` or `-XML` *`xml-file-name`*

specifies the name of the XML file, which is generated by the `mxexportddl` utility. The *`xml-file-name`* must be a valid OSS file name.

`-STATS ON |OFF`

specifies if the statistics of the tables needs to be imported. The default is ON.

`ON`

imports the statistics of the tables.

`OFF`

does not import the statistics of the tables. Only the metadata of the SQL/MX objects is imported.

`-CLEAR ON |OFF`

specifies if the `mximportddl` utility can overwrite the specified MAP file if it already exists and `-PREPAREMAP` is provided. The default is OFF.

`ON`

overwrites the specified MAP file if it already exists.

`OFF`

does not overwrite the specified MAP file. If the file already exists, the write operation fails.

`-KEEPDDL ON | OFF`

specifies whether the `mximportddl` utility should retain the target tables and its corresponding indexes if it already exists. The default is ON.

`ON`

imports the metadata of table and its indexes only if the target table does not exist. If the target table exists, physical statistics (index level, nonempty block count, and EOF) will not be imported regardless of the `KEEPSTATS` option value.

`OFF`

imports the metadata even if the target table already exists. The existing table and its indexes will be dropped, before creating the table. The existing statistics will also be deleted. The `KEEPDDL OFF` option will override the `KEEPSTATS ON` option.

`-KEEPGFN ON | OFF`

specifies whether the Guardian file names of table and index partitions must be retained by the `mximportddl` utility. The default is ON.

`ON`

imports the partitions with same Guardian file names as in the original table.

`OFF`

imports the partitions with SQL/MX generated Guardian file names.

`-KEEPSTATS ON | OFF`

specifies whether the `mximportddl` utility should retain the statistics of target tables if they already exist. The default is ON.

`ON`

imports the statistics only if the target table does not have statistics.

`OFF`

imports the statistics even if the target table already has statistics. The existing statistics will be deleted before importing the statistics.

`-LISTONLY ON | OFF`

lists the SQL/MX objects in the specified XML file. Object is not imported. The default is OFF.

`ON`

lists the objects in the XML file that matches the specified *`sqlmx-objects-spec-list`*. The specified SQL/MX objects are not imported and will not display the statistics details.

`OFF`

imports the specified SQL/MX objects.

`-SHOWDDL ON | OFF`

generates OSS files containing DDL information of dependency objects. The default is OFF.

`ON`

generates one or more OSS files containing DDL information of dependency objects. These files are created in addition to the objects that are normally imported.

`OFF`

does not generate OSS files containing DDL information of dependency objects.

Guidelines

- If `SHOWDDL ON` is specified, the `SHOWDDLLOC` option also needs to be specified.
- The DDL information can be used to manually create triggers, views, referential integrity constraints, and stored procedures after importing the

SQL/MX objects. The generated files are named as shown in the following table:

File Name	Purpose	Subordinate object of...
SHOWDDLn_RI_Constraints	Referential integrity	Table
SHOWDDLn_Stored_Procedures	Java stored procedures	Schema
SHOWDDLn_Triggers	Triggers	Table
SHOWDDLn_Views	Views	Schema

n starts at 0 and increments by one. A new file is created every time when the current file becomes full.

`-SHOWDDLLOC` *`oss-directory`*

places the files containing DDL information in the specified *`oss-directory`*. The *`oss-directory`* must be a valid OSS directory.

Guidelines

- Along with the `-SHOWDDLLOC` *`oss-directory`* option, the `SHOWDDL ON` option needs to be enabled.
- If the OSS directory already contains files with the same name as the files that are being generated, the original files are overwritten.

`-PREPAREMAP`

generates the MAP file from the provided XML file. No object will be imported. The generated MAP file can be used to map the parent object names and Guardian locations of the SQL/MX objects.

Considerations for mximportddl

You can edit the XML file manually using simple text editors. However, this method can be error-prone and is not recommended.

As the schema owner information in the XML file can be updated, other users can import the schema and its objects.

Remote catalogs must be registered manually before executing `mximportddl`. While importing, `mximportddl` will not register catalogs on the remote system.

`mximportddl` supports the RI actions CASCADE/SET NULL/SET DEFAULT in addition to NO ACTION and RESTRICT.

Supported by mximportddl

- Tables and associated objects such as indexes and partitions
- Constraints: check, not null, primary key, and unique

- Table statistics that includes physical statistics (index level, nonempty block count, and EOF), histograms, and histogram intervals
- Referential integrity constraints, views, triggers, and stored procedures
- For a table containing the IDENTITY column, the internal Sequence Generator attributes will also be stored in the XML file

Not Supported by mximportddl

- System defaults
- SQL/MP tables and aliases

Examples of mximportddl

- To import the catalog `cat1` and schema `cat2.sch1` and `cat2.sch2`:

  ```
  mximportddl -cat cat1 -sch cat2.sch1 cat2.sch2 -xml
  export.xml
  ```

- To import the catalog `cat1` without statistics:

  ```
  mximportddl -cat cat1 -xml export.xml -stats off
  ```

- To import all the SQL/MX objects existing in the XML file:

  ```
  mximportddl -xml export.xml
  ```

- To generate the MAP file template from the XML file:

  ```
  mximportddl -preparemap -xml export.xml -map export.map
  ```

  The sample MAP FILE is shown below:

```
[CATALOG-MAPPING]
CATALOG CAT1 = NCAT1;


[SCHEMA-MAPPING]
SCHEMA CAT2.SCH1 = NCAT3.NSCH;


[LOCATION-MAPPING]
LOCATION \NSK.$DATA1 = @CURRENTNODE.$DATA03;
LOCATION \NSK.$DATA2 = @CURRENTNODE.$DATA04;
LOCATION \NSK.$DATA3 = @CURRENTNODE.$DATA05;
LOCATION \NSK.$DATA4 = \DEV5.$DATA06;
```

  `@CURRENTNODE` will be replaced by the system name in which the `mximportddl` utility is running.

- To import the SQL/MX objects to a different target using map file:

  ```
  mximportddl -xml export.xml -map export.map -cat cat1 cat2
  ```

MXRPM tool

MXRPM is an OSS tool that reprocesses module files. This tool processes module files with version 3200 and above.

This tool helps to improve embedded SQL application performance by persisting the result of Similarity Check in the module files. When you move the module files from a test to a production system, use this tool to replace compile time objects and their specific attributes with run time objects and their specific attributes in the module file. When the application loads in the production environment, Similarity Checks are avoided for statements in the module files that are successfully reprocessed by the tool.

```
mxrpm   [ -log [ log-file ] ]
          -map map-file
        { -modlist modules-list-input-file | module-
        files-list-separated-by-spaces }
        | -help
```

`-map` *`map-file`*

`map-file` is an unstructured OSS file, which contains the mappings between compile time and runtime ANSI names for tables and views. If the absolute path for the file is not specified, the tool searches for the file in the current directory.

The compile time and runtime ANSI names specified in the *`map-file`* can contain only ISO88591 characters. A colon is used as a delimiter between the compile time and the runtime ANSI names. Any leading and trailing spaces are removed. Each mapping must be on a separate line. For more information, see Guidelines for map-file on page 5-76.

`-log` *`log-file`*

This tool logs the result of processing the module files in *`log-file`*. If the `-log` option is specified without the *`log-file`*, it logs the result to the console. For more information, see Guidelines for log-file on page 5-76.

`-modlist` *`modules-list-input-file`*

`modules-list-input-file` is an unstructured OSS file which contains the list of module files. This tool can process only OSS module files. Each module file must be specified on a separate line. For more information, see Guidelines for module-list-input-file on page 5-76.

`module-files-list-separated-by-spaces`

You can specify the modules on the command line with the module file names separated by spaces.

`-help`

provides information on how to use the tool.

Note. HP recommends that you take a backup of the module files before processing.

Guidelines for *map-file*

The guidelines to create a map file are:

- Specify a single mapping between a compile time and runtime table or view name. This tool stops processing and returns an error if it detects more than one mapping for the same compile time table or view.
- Specify three part ANSI names for compile time and runtime names.
- All references to the PROTOTYPE table or view will be mapped to the runtime table or view provided in the *`map-file`*.
- Mappings are not required for indexes.

Note. You must have read permission on the *map-file* and search permission on the directory.

Guidelines for *module-list-input-file*

The guidelines to create a *`module-list-input-file`* are:

- List each module file on a separate line.
- This tool does not process system module files. It processes the module files sequentially and stops if it finds a system module file.
- You must have read and write permissions on the module file.
- If the full path for the module file is not specified, this tool uses the following search sequence for the module file:
    - Local directory (from where you launch the tool)
    - User specified directories using the environment variables or Guardian DEFINES
    - Global module directory
- If the full path is specified and the file does not exist, the tool will not process the module file.

Guidelines for *log-file*

- If the absolute path is not specified, this tool searches for the file in the current directory. It creates a file if the file does not exist. If the file exists, this tool renames the existing file with an extension `.sqlmx.bak` and creates a new file. If the file with extension `.sqlmx.bak` already exists, this tool purges the file.

- To display logging information on the terminal, specify the `-log` option without a filename. If the option is not specified, only summary report is displayed. Do not use the wildcards in the log file name.

Note. You must have write and search permission on the directory to create a log file. If a log file exists, you must have write permission on the file.

Considerations

The following are considerations when running this tool:

- It supports only SQL/MX tables or views.
- It does not support SQL/MP tables or views. If a statement accesses SQL/MP tables or views, SQL/MP tables using MP aliases or both SQL/MP and SQL/MX objects, this tool skips the statement and does not update the redefinition timestamps or the ANSI names.
- It does not process SQL/MX queries using DEFINEs.
- It uses the existing Similarity Check criteria to determine if the compile time and runtime tables or views are similar. It does not override any existing Similarity Check criteria. If the Similarity Check fails for a statement, this tool does not update the redefinition timestamps for the statement. An automatic recompilation occurs for these statements if Similarity Check is enabled at runtime.
- If you compile the module files with Similarity Check disabled, this tool skips Similarity Check on statements in these module files.
- It processes both OSS and Guardian module files. You can provide an absolute path for the module file. If the absolute path is not specified, the module file is searched using the search criteria. For more information, see Guidelines for module-list-input-file on page 5-76. For Guardian modules, specify either the three-part module name as specified in the application or the fully qualified filename starting with /G.
- It creates a temporary module file in the location specified by TMPDIR environment variable. If TMPDIR is not set, this tool creates a temporary module file in the current directory. If both TMPDIR and current directories are Guardian locations, this tool returns an error. You must have the read and write permissions for these locations.

  If the temporary module file, which is composed by appending _sqlmx_temp to the module file name, already exists, this tool returns an error.

  It replaces the original module file with the temporary module file upon successful processing of the module file. You must have write permission on the module file directory.

  This tool purges the temporary module file if there is an error in processing or there are no changes to the original module file.

mxtool Utility

`mxtool` is an OSS command-line utility that performs various utility functions.

```
mxtool utility-operation
utility-operation is

  {  FIXUP fixup operation
     | GOAWAY goaway operation
     | HELP [help options]
     | INFO info operation
     | VERIFY verify operation
     | CLEANUP cleanup operation
}
help options is

  { ALL   | FIXUP   | GOAWAY   | INFO | VERIFY | CLEANUP }
```

`mxtool` *`utility-operation`*

executes this utility, which must appear in lowercase letters.

`utility-operation`

is the operation to be performed. It is not case-sensitive. *`utility-operation`* is one of:

FIXRCB Operation on page 5-7
mxtool FIXUP TABLE mycat.mysch.FIXUPtable -ru -d on page 5-12
FIXUP Operation on page 5-8
GOAWAY Operation on page 5-13
INFO Operation on page 5-53
VERIFY Operation on page 5-79
CLEANUP Operation on page 5-3

`HELP`
and
`HELP ALL`

display helpful information about the `mxtool` command-line options.

VERIFY Operation

[Considerations for VERIFY](#)
[Examples of VERIFY](#)

VERIFY is an OSS command-line utility run from `mxtool` that reports whether SQL/MX objects and programs are consistently described in file labels, resource forks, and metadata. Starting with SQL/MX Release 3.2, the VERIFY operation can produce a list of orphaned Guardian files. The orphaned Guardian files are existing Guardian files for which metadata information is not available in the SQLMX database.

```
mxtool VERIFY { object-option  | file-option | orphan-file-
option}

object-option is: ANSI name of the object

file option is: PART guardian-file-name

guardian-file-name is: [\node.]$volume.subvol.filename

orphan-file-option is:  -oo  wildcard-guardian-file-name
                       output


wildcard-guardian-file-name is:

Guardian file name, which includes the wild card character,*.

output is: -f=OSS-file-name
```

`object-option`

specifies that VERIFY should be performed against all partitions and dependent indexes of a table.

`catalog.schema.object`

is the fully qualified ANSI name of a table. If any of the three parts of the name is an SQL/MX reserved word, you must delimit it by enclosing it in double quotes. Such delimited parts are case-sensitive. For example: `cat.sch."join"`.

VERIFY displays inconsistencies to the standard output file.

`file-option`

specifies that VERIFY should be performed against a file. Only a table object can be specified.

`orphan-file-option`

specifies that VERIFY must report the name(s) of an orphaned Guardian file or set of files.

guardian-file-name

is a single Guardian physical file name to be verified. The Guardian file name specification can contain a node, volume, subvolume, and a file name. The node name is optional.

wildcard-guardian-file-name

allows wild card character "*" as part of the Guardian file name. For example,

```
\dmr15.$*.*.*
```

```
\dmr15.$data*.ZSDV1*.PART*
```

OSS-file-name

the name of an OSS file. When the `-oo` option is used, VERIFY writes the names of orphaned Guardian files to this file, one orphaned file name per line.

[*node*.]$*volume.subvol.filename*

is the Guardian qualified file name that describes the partition that is being queried.

In this four-part name, *node* is the name of a node of a NonStop server, *$volume* is the name of a disk volume, *subvol* is the name of a subvolume, and *filename* is the name of an SQL/MX table or view.

If the name contains special characters such as "\" or "$", you must precede these characters with a backslash (\), or you can enclose the entire four-part name in single quotes. For example:

`\\node2.\$data3.sales.mytable` or `'\node2.$data3.sales.mytable'`.

If you do not specify *node*, the default is the Guardian system named in your =_DEFAULTS define.

You can find the Guardian (physical) file name by using the SHOWLABEL command. For example:

```
SHOWLABEL CAT.SCH.T1, DETAIL;
```

For more information, see SHOWLABEL Command on page 4-99.

Considerations for VERIFY

When the object option or file option is used, VERIFY checks for inconsistencies between information stored in the metadata and information stored in labels, including:

- ANSI name
- ANSI namespace
- Partition map, including:
    - Number of partitions
    - First key values
    - Physical locations
- Version information

- Number of indexes
- Partition name
- For dependent indexes, similar checks are made for inconsistencies, including:
  - ANSI name
  - ANSI namespace
  - Partition map
  - Version information
- Constraint information.

  Constraint information stored in the label is used during similarity checking to see if the current plan can be executed or needs to be recompiled. The number of constraints defined must be the same. In addition, the constraint text and disabled attribute must match. Only droppable check constraint information is verified.
- Redefinition time
- Extent sizes (primary extent size, secondary extent size, maximum number of extents)
- Audit flag
- Corrupt flag

If a table has offline partitions or unpopulated indexes defined in metadata, they are noted in the output. If a table has both offline partitions and unpopulated indexes defined in metadata, they will be ignored.

When the `-oo` option is used, VERIFY searches the set of Guardian files that match the specified *`file-name-template.`* For each matching file, VERIFY checks if the corresponding SQL/MX metadata exists. If not, that Guardian file is considered orphaned, and its name is written to the specified output file.

After VERIFY has completed the search for orphaned Guardian files, the resulting output file can be used as input to CLEANUP.

Security Considerations

- VERIFY does not check privilege information.
- VERIFY obtains read-only locks on metadata while verifying an object. Other operations that read metadata can run concurrently. Operations that change metadata or labels such as DDL, partition management, PURGEDATA, and UPDATE STATISTICS statements cannot run concurrently.
- To verify some objects NonStop SQL/MX might need to access remote systems. The remote system must be available and you must have privileges to view information on it.
- If VERIFY tries to access objects that have a schema version that is greater than the SQL/MX software version (MXV) of the local node, you receive a versioning error.

Examples of VERIFY

- This example shows the creation of a table:

```
CREATE TABLE payroll.dec2000.hourly

  C1   INT NO DEFAULT -- NOT NULL NOT DROPPABLE
, C2   INT DEFAULT NULL
, C3   CHAR(10) CHARACTER SET ISO88591 COLLATE DEFAULT
DEFAULT NULL
, CONSTRAINT payroll.dec2000.hourly_392165858_3314 PRIMARY
KEY (C1 ASC)
    NOT DROPPABLE
, CONSTRAINT payroll.dec2000.hourly_328843858_3314 CHECK
    (payroll.dec2000.hourly.C1 IS NOT NULL) NOT DROPPABLE
)
LOCATION \SQUAW.$DATA08.ZSDVVVVV.T1000100
NAME SQUAW_DATA08_ZSDVVVVV_T1000100
ATTRIBUTES EXTENT (64, 512), MAXEXTENTS 400
PARTITION
(
  ADD FIRST KEY (1000)
    LOCATION \SQUAW.$DATA08.ZSDVVVVV.T1000200
    NAME SQUAW_DATA08_ZSDVVVVV_T1000200
    EXTENT (16, 512) MAXEXTENTS 100
, ADD FIRST KEY (2000)
    LOCATION \SQUAW.$DATA08.ZSDVVVVV.T1000300
    NAME SQUAW_DATA08_ZSDVVVVV_T1000300
    EXTENT (16, 512) MAXEXTENTS 200
, ADD FIRST KEY (3000)
    LOCATION \SQUAW.$DATA08.ZSDVVVVV.T1000400
    NAME SQUAW_DATA08_ZSDVVVVV_T1000400
    EXTENT (16, 1024) MAXEXTENTS 300
)
STORE BY (C1 ASC);

CREATE INDEX VERIFY_T1_NDX2 ON payroll.dec2000.hourly
  (
    C3 ASC
  )
  LOCATION \SQUAW.$DATA08.ZSDVVVVV.T1X20100
  NAME SQUAW_DATA08_ZSDVVVVV_T1X20100
  PARTITION
  (
    ADD FIRST KEY (_ISO88591'a1')
      LOCATION \SQUAW.$DATA08.ZSDVVVVV.T1X20200
      NAME SQUAW_DATA08_ZSDVVVVV_T1X20200
      EXTENT (16, 128) MAXEXTENTS 200
  , ADD FIRST KEY (_ISO88591'a2')
      LOCATION \SQUAW.$DATA08.ZSDVVVVV.T1000300
      NAME SQUAW_DATA08_ZSDVVVVV_T1000300
      EXTENT (16, 512) MAXEXTENTS 200
  , ADD FIRST KEY (3000)
      LOCATION \SQUAW.$DATA08.ZSDVVVVV.T1000400
      NAME SQUAW_DATA08_ZSDVVVVV_T1000400
```

```
        EXTENT (16, 1024) MAXEXTENTS 300
    )
    STORE BY (C1 ASC);

ALTER TABLE payroll.dec2000.hourly
  ADD CONSTRAINT payroll.dec2000.hourly_chk1 CHECK
      (payroll.dec2000.hourly.C2 > 0) DROPPABLE ;
```

- This example shows the VERIFY operation run against table `payroll.dec2000.hourly`, without options. VERIFY defaults to the -a option:

```
mxtool VERIFY payroll.dec2000.hourly

NonStop SQL/MX MXTOOL Utility 2.0
(c) Copyright 2003 Hewlett-Packard Development Company, LP.
All Rights Reserved.

Verifying table:  payroll.dec2000.hourly

 Verifying label for partition :
\SQUAW.$DATA08.ZSDVVVVV.T1000100

 Verifying resource fork for partition:
\SQUAW.$DATA08.ZSDVVVVV.T1000100
 Verifying constraints:  \SQUAW.$DATA08.ZSDVVVVV.T1000100
 Verifying Partition Map:  \SQUAW.$DATA08.ZSDVVVVV.T1000100
 Verifying Index Map :  \SQUAW.$DATA08.ZSDVVVVV.T1000100

 Verifying label for partition :
\SQUAW.$DATA08.ZSDVVVVV.T1000200

 Verifying resource fork for partition:
\SQUAW.$DATA08.ZSDVVVVV.T1000200
 Verifying constraints:  \SQUAW.$DATA08.ZSDVVVVV.T1000200
 Verifying Partition Map:  \SQUAW.$DATA08.ZSDVVVVV.T1000200
 Verifying Index Map :  \SQUAW.$DATA08.ZSDVVVVV.T1000200

 Verifying label for partition :
\SQUAW.$DATA08.ZSDVVVVV.T1000300

 Verifying resource fork for partition:
\SQUAW.$DATA08.ZSDVVVVV.T1000300
 Verifying constraints:  \SQUAW.$DATA08.ZSDVVVVV.T1000300
 Verifying Partition Map:  \SQUAW.$DATA08.ZSDVVVVV.T1000300
 Verifying Index Map :  \SQUAW.$DATA08.ZSDVVVVV.T1000300

 Verifying label for partition :
\SQUAW.$DATA08.ZSDVVVVV.T1000400

 Verifying resource fork for partition:
\SQUAW.$DATA08.ZSDVVVVV.T1000400
 Verifying constraints:  \SQUAW.$DATA08.ZSDVVVVV.T1000400
 Verifying Partition Map:  \SQUAW.$DATA08.ZSDVVVVV.T1000400
 Verifying Index Map :  \SQUAW.$DATA08.ZSDVVVVV.T1000400
```

```
Verifying index: payroll.dec2000.VERIFY_T1_NDX2

 Verifying label for partition :
\SQUAW.$DATA08.ZSDVVVVV.T1X20100

 Verifying resource fork for partition:
\SQUAW.$DATA08.ZSDVVVVV.T1X20100
 Verifying Partition Map:  \SQUAW.$DATA08.ZSDVVVVV.T1X20100

 Verifying label for partition :
\SQUAW.$DATA08.ZSDVVVVV.T1X20200

 Verifying resource fork for partition:
\SQUAW.$DATA08.ZSDVVVVV.T1X20200
 Verifying Partition Map:  \SQUAW.$DATA08.ZSDVVVVV.T1X20200

 Verifying label for partition :
\SQUAW.$DATA08.ZSDVVVVV.T1X20300

 Verifying resource fork for partition:
\SQUAW.$DATA08.ZSDVVVVV.T1X20300
 Verifying Partition Map:  \SQUAW.$DATA08.ZSDVVVVV.T1X20300

 Verifying label for partition :
\SQUAW.$DATA08.ZSDVVVVV.T1X20400

 Verifying resource fork for partition:
\SQUAW.$DATA08.ZSDVVVVV.T1X20400
 Verifying Partition Map:  \SQUAW.$DATA08.ZSDVVVVV.T1X20400

Object verification complete for : payroll.dec2000.hourly
```

6 SQL/MX Language Elements

NonStop SQL/MX language elements, which include data types, expressions, functions, identifiers, literals, and predicates, occur within the syntax of SQL/MX statements and MXCI commands. The statement and command topics support the syntactical and semantic descriptions of the language elements in this section.

This section describes:

Catalogs

SQL/MX Catalogs

An SQL/MX catalog is a named logical object that contains descriptions of a set of schemas. You can access SQL/MX objects with the three-part name of the actual object.

The ANSI SQL:1999 catalog name is an SQL identifier. In SQL/MX Release 2.x, ANSI catalogs do not have any physical representation, nor do they have a physical relationship to SQL/MP catalogs.

A catalog is owned by the user ID that created it, though catalog ownership does not imply authorization over schemas or objects in that catalog, and any user can drop an empty catalog, regardless who the catalog owner is. Each of the schemas described in a catalog has an owner. A catalog can contain multiple schemas, each possibly owned by a different user. A catalog cannot contain other catalogs. Any user on a node can create a catalog on that node. The catalog’s owner has the authority to register and unregister the catalog.

An SQL/MX catalog name can be up to 128 characters and is location-independent.

SQL/MP Catalogs

An SQL/MP catalog is a set of tables and indexes that describe SQL objects. Tables in the set are called catalog tables and NonStop SQL/MP creates them, along with their indexes, when you execute a CREATE CATALOG statement. Each catalog resides on its own Guardian subvolume, and the name of that subvolume is also the name of the catalog.

A catalog name has the form: `[\`*`node`*`.][$`*`volume`*`.]`*`subvol`*

Each node on which NonStop SQL/MP is used has one special catalog called the system catalog and might have many other catalogs. Each table, view, index, partition, collation, or catalog table located on a node must be described in a catalog on the same node. For more information, see the *SQL/MP Reference Manual*.

See SET CATALOG Statement on page 2-366 and Object Naming on page 10-60.

Character Sets

When you run the `InstallSqlmx` script during NonStop SQL/MX installation, you specify the NATIONAL_CHARSET attribute to select a default NCHAR character set of UCS2, ISO88591, KANJI, or KSC5601. If you do not specify a character set, the default is UCS2. Once the default is set, you cannot change it. For more information about setting this default, see the instructions for installing NonStop SQL/MX in the *SQL/MX Installation and Management Guide*.

Note. KANJI and KSC5601 are valid character sets for SQL/MP tables but not SQL/MX tables. If you attempt to create an SQL/MX table with KANJI, KSC5601, or other unsupported character sets, you get an SQL error and the operation fails.

After you have set the character set default, when you create SQL/MX tables, NCHAR data type fields use this character set as the default.

Within programs, NonStop SQL/MX allows you to associate one of these character sets with a literal or host variable:

ISO88591	Default single-byte 8-bit character set for character data types, which supports English and other Western European languages.
UCS2	Double-byte Unicode character set in UTF16 big-endian encoding. All Basic Multilingual Plan (BMP) characters are included. Surrogate characters are treated as two double-byte characters.
KANJI	Double-byte character set widely used on Japanese mainframes. It is a subset of Shift JIS (the double character portion). Its encoding is big-endian.
KSC5601	Double-byte character set required on systems used by government and banking within Korea. Its encoding is big-endian.

KANJI and KSC5601 are valid only for SQL/MP tables.

For more information about defining character data, see the guidelines for creating an SQL/MX database in the *SQL/MX Installation and Management Guide*.

Restrictions on Using Character Set Data

For SQL/MX tables, only ISO88591 characters are allowed in these fields:

ISO88591 Field	Where Found
BY partitioning-column	CREATE INDEX, CREATE TABLE statements
FIRST KEY values	CREATE TABLE, CREATE INDEX statements; MODIFY TABLE utility.
CHECK constraint text	CREATE TABLE and ALTER TABLE statements
Column HEADING text	CREATE TABLE and ALTER TABLE statement

ISO88591 Field	Where Found
View text	CREATE VIEW statement
$volume specification	CREATE CATALOG, CREATE TABLE, CREATE INDEX, CREATE PROCEDURE, CREATE VIEW, DUP, and PURGEDATA statements; MODIFY and RESTORE utilities
SQL/MX names	Names of catalogs, columns, constraints, indexes, schemas, stored procedures, tables, and views

In addition, user data fields in SQL/MX tables must use either ISO88591 or UCS2. KANJI and KSC5601 are not allowed.

In SQL/MP tables, a character data type has an associated character set and collation that can be implicitly or explicitly specified. Internally, the ISO88591 character set is implemented as an 8-bit data type, while the UCS2, KANJI, and KSC5601 character sets are implemented as 16-bit data types. The CHAR data type can be associated with any of the character sets. The NCHAR data type is typically associated with the UCS2 character set.

You can insert into and update NCHAR columns in an SQL/MP table. See Character String Literals on page 6-64. You can query SQL/MP tables that have columns associated with the KANJI or KSC5601 character sets.

Collations

A collation is an object that contains rules for a collating sequence (the sequence in which characters are ordered for sorting), case, and character class and character string equivalence.

Every character set has a collation. See Character Sets on page 6-4. To be compared, character strings must be from the same character set. When two strings are compared, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks to have the same length as the longer string.

You create an SQL/MP collation with the SQL/MP CREATE COLLATION statement. A collation name must be a Guardian name. For more information, see the *SQL/MP Reference Manual*.

You cannot use SQL/MP collations on SQL/MX tables and you cannot create collations for SQL/MX tables. SQL/MX Release 2.x supports only the DEFAULT collation. DEFAULT is based on binary ordering and is the default collating sequence for CHAR and NCHAR data types.

Binary collation is a collating sequence based on binary ordering. A binary collation comparison of two equal length strings, *s1* and *s2*, compares the values of the corresponding characters of *s1* and *s2* until it finds a difference. If a difference is found and the differing character value of *s1* is less than that of *s2*, *s1* is considered to come before *s2*. If there is no difference, *s1* is considered equal to *s2*. Otherwise, *s1* comes after *s2*. If the two strings are not equal in length, the shorter one is padded with spaces in the corresponding character set.

Comparison of two identical strings associated with the same character set will always be evaluated as equal by the DEFAULT collation. However, the DEFAULT collation does not necessarily yield sorting orders that are culturally correct.

Columns

Examples of Derived Column Names

A column is a vertical component of a table and is the relational representation of a field in a record. A column contains one data value for each row of the table.

A column value is the smallest unit of data that can be selected from or updated in a table. Each column has a name that is an SQL identifier and is unique within the table or view that contains the column.

Column References

A qualified column name, or column reference, is a column name qualified by the name of the table or view to which the column belongs, or by a correlation name.

If a query refers to columns that have the same name but belong to different tables, you must use a qualified column name to refer to the columns within the query. You must also refer to a column by a qualified column name if you join a table with itself within a query to compare one row of the table with other rows in the same table.

The syntax of a column reference or qualified column name is:

```
{table-name | view-name | correlation-name}.column-name
```

If you define a correlation name for a table in the FROM clause of a statement, you must use that correlation name if you need to qualify the column name within the statement.

If you do not define an explicit correlation name in the FROM clause, you can qualify the column name with the name of the table or view that contains the column. See Correlation Names on page 6-11.

Derived Column Names

A derived column is an SQL value expression that appears as an item in the select list of a SELECT statement. An explicit name for a derived column is an SQL identifier associated with the derived column. The syntax of a derived column name is:

```
column-expression [[AS] column-name]
```

The column expression can simply be a column reference. The expression is optionally followed by the AS keyword and the name of the derived column.

If you do not assign a name to derived columns, the headings for unnamed columns in query result tables appear as (EXPR). Use the AS clause to assign names that are meaningful to you, which is important if you have more than one derived column in your select list.

Column Default Settings

You can define specific default settings for columns when the table is created. The CREATE TABLE statement defines the default settings for columns within tables. The default setting for a column is the value inserted in a row when an INSERT statement omits a value for a particular column.

Examples of Derived Column Names

- These two examples show how to use names for derived columns.

  The first example shows (EXPR) as the column heading of the SELECT result table:

```
SELECT AVG (salary)
FROM persnl.employee;
(EXPR)
----------------
        49441.52

--- 1 row(s) selected.
```

  The second example shows AVERAGE SALARY as the column heading:

```
SELECT AVG (salary) AS "AVERAGE SALARY"
FROM persnl.employee;
"AVERAGE SALARY"
----------------
        49441.52

--- 1 row(s) selected.
```

Constraints

An SQL/MX constraint is an object that protects the integrity of data in a table by specifying a condition that all the values in a particular column or set of columns of the table must satisfy.

NonStop SQL/MX enforces these constraints on SQL/MP and SQL/MX tables:

CHECK	Column or table constraint specifying a condition must be satisfied for each row in the table. For SQL/MX tables, check constraints cannot contain non-ISO88591 string literals.
NOT NULL	Column constraint specifying the column cannot contain nulls.
PRIMARY KEY	Column or table constraint specifying the column or set of columns as the primary key for the table.
REFERENTIAL INTEGRITY	Column or table constraint specifying a referential constraint: a column or set of columns in the table can contain only values matching those in a column or set of columns in the referenced table. This type of constraint is also called a *references column constraint*. (SQL/MX tables only.)
UNIQUE	Column or table constraint specifying the column or set of columns cannot contain more than one occurrence of the same non-null value or set of values. (SQL/MX tables only.)

Creating, Adding, and Dropping Constraints on SQL/MX Tables

To create constraints on an SQL/MX table when you create the table, use the CHECK, NOT NULL, PRIMARY KEY, [FOREIGN KEY] REFERENCES, or UNIQUE clauses of the CREATE TABLE statement.

To add or drop constraints on an existing table, use the CHECK, PRIMARY KEY, [FOREIGN KEY] REFERENCES, or UNIQUE clauses of the ALTER TABLE statement. You will receive an error if rows that already exist in the table violate that constraint.

You can define constraints either on a single column (column constraint) or on a set of columns (table constraint). You can create a NOT NULL column constraint by using CREATE TABLE and drop NOT NULL by using ALTER TABLE. All other constraints can be added or dropped by using ALTER TABLE.

You can specify a NOT NULL or PRIMARY KEY constraint as NOT DROPPABLE at table creation time. NonStop SQL/MX implements these constraints more efficiently if they are specified as NOT DROPPABLE. For performance reasons, all NOT NULL NOT DROPPABLE constraints for a table are replaced by a single CHECK constraint that enforces the entire set.

For more information on SQL/MX commands, see CREATE TABLE Statement on page 2-107 and ALTER TABLE Statement on page 2-19.

Constraint Names

When you create a constraint, you can either specify a name for it or allow a name to be generated by NonStop SQL/MX. You can optionally specify both column and table constraint names. Constraint names are three-part logical names. Constraints have their own namespace within a schema, so a constraint name can have the same name as a table, index, or view. However, no two constraints in a schema can have the same name.

The name you specify can be fully qualified or not. If you specify the catalog or schema parts of the name, they must match those parts of the affected table and must be unique among constraint names in that schema. If you omit the catalog or schema portion of the name you specify, NonStop SQL/MX expands the name by using the catalog and schema for the table.

If you do not specify a constraint name, NonStop SQL/MX constructs an SQL identifier as the name for the constraint and qualifies it with the catalog and schema of the table. The identifier consists of the table name concatenated with a system-generated unique identifier. Use the SHOWDDL statement to display this generated constraint name.

Restrictions on Publish/Subscribe

Embedded update and embedded delete statements are not allowed on tables with referential integrity constraints.

Creating and Dropping Constraints on SQL/MP Tables

To create check constraints on an SQL/MP table, use the SQL/MP CREATE CONSTRAINT statement when you create the table. To drop constraints on an SQL/MP table, use the SQL/MP DROP statement. A constraint name is an SQL identifier.

For more information on SQL/MP commands, see the *SQL/MP Reference Manual*.

Correlation Names

A correlation name is a name you can associate with a table reference that is a table, view, or subquery in a SELECT statement to:

- Distinguish a table or view from another table or view referred to in a statement
- Distinguish different uses of the same table
- Make the query shorter

A correlation name can be explicit or implicit.

Explicit Correlation Names

An explicit correlation name for a table reference is an SQL identifier associated with the table reference in the FROM clause of a SELECT statement. The correlation name must be unique within the FROM clause. For more information about the FROM clause, table references, and correlation names, see SELECT Statement on page 2-330.

The syntax of a correlation name for the different forms of a table reference within a FROM clause is the same:

{*table* | *view* | (*query-expression*)} [AS]*correlation-name*

A table or view is optionally followed by the AS keyword and the correlation name. A derived table, resulting from the evaluation of a query expression, must be followed by the AS keyword and the correlation name. An explicit correlation name is known only to the statement in which you define it. You can use the same identifier as a correlation name in another statement.

Implicit Correlation Names

A table or view reference that has no explicit correlation name has an implicit correlation name. The implicit correlation name is the table or view name qualified with the catalog and schema names.

You cannot use an implicit correlation name for a reference that has an explicit correlation name within the statement.

Examples of Correlation Names

- This query refers to two tables (ORDERS and CUSTOMER) that contain columns named CUSTNUM. In the WHERE clause, one column reference is qualified by an implicit correlation name (ORDERS) and the other by an explicit correlation name (C):

```
SELECT ordernum, custname
FROM orders, customer c
WHERE orders.custnum = c.custnum
  AND orders.custnum = 543;
```

Database Objects

A database object is an SQL entity that exists in a namespace, maps to a Guardian file in most cases, and is registered in the system catalog. SQL/MX Release 2.x includes SQL/MX objects. SQL/MX DML statements can access both SQL/MX and SQL/MP objects. The subsections listed below describe these SQL/MX objects.

Collations
Constraints
Indexes
Partitions
SQL/MP Aliases
Stored Procedures
Tables
Triggers
Views

Ownership

In SQL/MX Release 2.x, the creator of a schema owns all the objects defined in the schema. In addition, NonStop SQL/MX allows the super ID to act as the owner of any object. In addition, you can use the GRANT and REVOKE statements to grant access privileges for a table or view to specified users.

For more information, see Security on page 1-5, EXPLAIN Statement on page 2-208, and REVOKE Statement on page 2-317. For more information on privileges on tables and views, see CREATE TABLE Statement on page 2-107, ALTER TABLE Statement on page 2-19, and CREATE VIEW Statement on page 2-154.

Database Object Names

SQL/MX DML statements can refer to SQL/MX database objects and SQL/MP database objects. To refer to a database object in a statement, use an appropriate database object name. For more information on the types of database objects, see Database Objects on page 6-12.

Logical Names for SQL/MX Objects

You can refer to an SQL/MX table, stored procedure, or view by using a three-part logical name, also called an ANSI name:

`catalog-name.schema-name.object-name`

In this three-part name, *catalog-name* is the name of the catalog, *schema-name* is the name of the schema, and *object-name* is the simple name of the table, stored procedure, or view. Each of the parts is an SQL identifier. See Identifiers on page 6-56. The NAMETYPE attribute defaults to ANSI, allowing you to use logical names of SQL/MX objects.

NonStop SQL/MX automatically qualifies an object name with the current default catalog and schema name unless you explicitly specify catalog and schema names with the object name. A two-part name *schema-name.object-name* is qualified implicitly with the current default catalog. A one-part name *object-name* is qualified implicitly with the default schema and catalog.

You can qualify a column name in an SQL/MX statement by using a three-part, two-part or one-part object name, or a correlation name.

For more information about the default catalog and schema, and the NAMETYPE attribute, see Object Naming on page 10-60.

Physical Names for SQL/MP Objects

Physical names of tables and views are qualified with the system node, volume, and subvolume names. SQL/MP tables and views are created with Guardian physical names of the form:

`[\node.][[$volume.]subvol.]filename`

In this four-part name, *\node* is the name of a node on a NonStop system, *$volume* is the name of a disk volume, *subvol* is the name of a subvolume, and *filename* is the name of a Guardian disk file or the name of an SQL/MP table or view.

You can choose to use physical names to refer to SQL/MP tables and views by setting the NAMETYPE attribute to NSK. If the NAMETYPE is NSK, NonStop SQL/MX automatically qualifies a physical table or view name with the current default node, volume, and subvolume names unless you explicitly specify these names with the object name.

In SQL/MX releases earlier than SQL/MX Release 2.x, if NAMETYPE was NSK and you did not use a correlation name, SQL/MX used the table part of the name as the correlation name. This behavior was similar to that of NonStop SQL/MP. In SQL/MX Release 2.x, if you do not use a correlation name, SQL/MX uses the default volume and subvolume to qualify the name. If the file does not exist, NonStop SQL/MX returns an error, which is ANSI-compliant behavior.

For more information about the default node, volume and subvolume names, and the NAMETYPE attribute, see Object Naming on page 10-60.

Logical Names for SQL/MP Objects

You can refer to an SQL/MP table or view by using the three-part logical name of an SQL/MP alias:

catalog-name.schema-name.object-name

In this three-part name, *catalog-name* is the name of the catalog, *schema-name* is the name of the schema, and *object-name* is the simple name of the table or view. Each of the parts is an SQL identifier. See Identifiers on page 6-56. The NAMETYPE attribute defaults to ANSI, allowing you to use logical names of SQL/MP aliases for SQL/MP objects.

NonStop SQL/MX automatically qualifies an object name with the current default catalog and schema name unless you explicitly specify catalog and schema names with the object name. A two-part name *schema-name.object-name* is qualified implicitly with the current default catalog. A one-part name *object-name* is qualified implicitly with the default schema and catalog.

If the NAMETYPE is ANSI, you can qualify a column name in an SQL/MX statement by using a three-part, two-part, or one-part object name, or a correlation name.

For more information about the default catalog and schema, and the NAMETYPE attribute, see Object Naming on page 10-60. For more information on assigning logical names to SQL/MP tables or views, see SQL/MP Aliases on page 6-112.

DEFINE Names for SQL/MP Objects

You can use DEFINE names as logical names for SQL/MP tables, views, or partitions in DML statements. When NonStop SQL/MX compiles such statements, it replaces the DEFINE name in the statement with the associated Guardian physical name. DEFINE names can be created within MXCI or can be inherited from the TACL process or the OSS shell.

You cannot use DEFINE names to refer to SQL/MX tables, views, partitions, or stored procedures.

The advantages of using DEFINEs rather than Guardian physical names are:

- DEFINE names are easier to understand than Guardian names.

  For example, the name =CUSTOMERS is simpler than the physical name \SYS.$VOL2.SALES.CSTMERS. See ADD DEFINE Command on page 4-4.

- DEFINE names provide location independence.

  If you use DEFINE names, you can change the physical file location without changing the SQL statement. See ALTER DEFINE Command on page 4-6. For more information on DEFINEs and late name resolution, see the *SQL/MX Programming Manual for C and COBOL*.

SQL/MX Object Namespaces

SQL/MX objects are organized in a hierarchical manner. Database objects exist in schemas, which are themselves contained in catalogs. Catalogs are collections of schemas. Schema names must be unique within a given catalog.

Multiple objects with the same name can exist provided that each belongs to a different namespace. NonStop SQL/MX supports these namespaces:

Namespace	Description
CN	Constraint
IX	Index
LK	Lock
MD	Module
TA	Table value object (table, view, stored procedure, MP Alias)
TR	Trigger
TT	Trigger temporary table

Objects in one schema can refer to objects in a different schema. Objects of a given namespace are required to have unique names within a given schema.

Considerations for Database Object Names

OBJECTS Table

The OBJECTS table is created at SQL/MX installation time and is used to store mappings from logical object names to physical Guardian locations. See OBJECTS Table on page 10-22.

You can use the CREATE SQLMP ALIAS command within your application to create the needed mappings from logical to physical names. This command has the form:

```
CREATE SQLMP ALIAS catalog-name.schema-name.table-name
                  [\node.]$volume.subvol.filename
```

When this command is executed, a mapping is inserted as a row in the OBJECTS table. SQL/MP aliases are simulated ANSI names that represent the underlying Guardian physical names of SQL/MP objects. True ANSI names do not exist for SQL/MP objects.

See DELETE Statement on page 2-162.

Mixing Name Types

- In a single SQL statement, tables or views can use ANSI logical names or Guardian physical names. You can combine these two name types in the same DML statement. For example:

```
SELECT salary FROM samdbcat.persnl.employee
WHERE \mysys.$samdb.persnl.empnum IN
  (SELECT mgr FROM \mysys.$samdb.persnl.dept);

INSERT INTO \mysys.$samdb.persnl.new_emps
  (SELECT * FROM samdbcat.persnl.employee);

SET NAMETYPE ANSI;
SET SCHEMA samdbcat.sales;
UPDATE odetail
  SET unit_price = unit_price * 10
  WHERE partnum IN
    (SELECT partnum FROM \mysys.$samdb.sales.parts);
```

- You can use DEFINE names and ANSI logical names in the same DML statement:

```
SELECT * FROM =parts p, samdbcat.sales.odetail o
  WHERE p.partnum = o.partnum;
```

- You can use DEFINE names and Guardian physical names in the same DML statement:

```
SELECT * FROM =parts p, \mysys.$samdb.sales.odetail o
  WHERE p.partnum = o.partnum;
```

Default Name Types

If the table or view names are partial names, they are fully qualified according to the rules of the current NAMETYPE attribute. The fully qualified names are either all ANSI logical names or all Guardian physical names. For more information, see Object Naming on page 10-60.

Data Types

SQL/MX data types are either character, datetime, interval, or numeric (exact or approximate):

Character String Data Types on page 6-22	Fixed-length and variable-length character data types.
Datetime Data Types on page 6-25	DATE, TIME, and TIMESTAMP data types.
Interval Data Types on page 6-31	Year-month intervals (years and months) and day-time intervals (days, hours, minutes, seconds, and fractions of a second).
Numeric Data Types on page 6-34	Exact and approximate numeric data types.

Each column in a table is associated with a data type. You can use the CAST expression to convert data to the data type that you specify. For more information, see CAST Expression on page 8-20.

Comparable and Compatible Data Types

Two data types are comparable if a value of one data type can be compared to a value of the other data type.

Two data types are compatible if a value of one data type can be assigned to a column of the other data type, and if columns of the two data types can be combined using arithmetic operations. Compatible data types are also comparable.

Assignment and comparison are the basic operations of NonStop SQL/MX. Assignment operations are performed during the execution of INSERT and UPDATE statements. Comparison operations are performed during the execution of statements that include predicates, aggregate (or set) functions, and GROUP BY, HAVING, and ORDER BY clauses.

The basic rule for both assignment and comparison is that the operands have compatible data types. For assignment operations, a further restriction is that null cannot be assigned to a column that has been defined as NOT NULL. Data types with different character sets cannot be compared.

Character Data Types

Values of fixed and variable length character data types of the same character set are all character strings and are all mutually comparable and mutually assignable.

When two strings are compared, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks to have the same length as the longer string.

Datetime Data Types

Values of type datetime are mutually comparable and mutually assignable only if the types have the same datetime fields. A DATE, TIME, or TIMESTAMP value can be compared with another value only if the other value has the same data type.

All comparisons are chronological. For example, this predicate is true:

```
TIMESTAMP '1997-09-28 00:00:00' >
  TIMESTAMP '1997-06-26 00:00:00'
```

Interval Data Types

Values of type INTERVAL are mutually comparable and mutually assignable only if the types are either both year-month intervals or both day-time intervals.

For example, this predicate is true:

```
INTERVAL '02-01' YEAR TO MONTH > INTERVAL '00-01' YEAR TO MONTH
```

The field components of the INTERVAL do not have to be the same. For example, this predicate is also true:

```
INTERVAL '02-01' YEAR TO MONTH > INTERVAL '01' YEAR
```

Numeric Data Types

Values of the approximate data types FLOAT, REAL, and DOUBLE PRECISION, and values of the exact data types NUMERIC, DECIMAL, INTEGER, SMALLINT, and LARGEINT, are all numbers and are all mutually comparable and mutually assignable.

When an approximate data type value is assigned to a column with exact data type, rounding might occur, and the fractional part might be truncated. When an exact data type value is assigned to a column with approximate data type, the result might not be identical to the original number.

When two numbers are compared, the comparison is made with a temporary copy of one of the numbers, according to defined rules of conversion. For example, if one number is INTEGER and the other is DECIMAL, the comparison is made with a temporary copy of the integer converted to a decimal.

Extended NUMERIC Precision

SQL/MX provides support for extended NUMERIC precision data type. Extended NUMERIC is either a signed numeric value with precision greater than 18 or an unsigned numeric value with precision greater than 9.

Note. Dynamic SQL programs must convert the extended NUMERIC precision data type to other compatible data types, such as CHAR.

Considerations for Extended NUMERIC Precision Data Type

- Supported in all DDL and DML statements where an ordinary NUMERIC data type is supported.
    - MX tables only
- Supported from the MXCI, ODBC, JDBC T2 and T4 interfaces.
- Does not support for host variable declarations in embedded programs.
- CAST function allows conversion between an ordinary NUMERIC and extended NUMERIC precision data type.

  To convert a signed extended NUMERIC data type to CHAR data type, the required length of the CHAR host variable is p + 3, where p is the precision of the extended NUMERIC data type. Three extra bytes are for the sign, decimal point, and the null terminator. For unsigned extended NUMERIC data type, the required length is p + 2.
- Implemented in software (versus hardware for ordinary numeric data type) and therefore is CPU intensive.
- Supported in arithmetic operations of addition, subtraction, multiplication, division, and exponentiation.
    - Supported as a parameter in the following scalar functions:

ABS	ACOS	ASIN
ATAN	ATAN2	AVG
CEILING	COS	COSH
COUNT	DEGREES	DIFF1
DIFF2	EXP	FLOOR
HASHPARTFUNC	INSERT	LASTNOTNULL
LEFT	LOG	LOG10
LPAD	MAX	MIN
MOVINGAVG	MOVINGCOUNT	MOVINGMAX
MOVINGMIN	MOVINGSTDDEV	MOVINGSUM
MOVINGVARIANCE	OFFSET	POWER
RADIANS	REPEAT	RIGHT
ROWS SINCE	RPAD	RUNNINGAVG
RUNNINGCOUNT	RUNNINGMAX	RUNNINGMIN
RUNNINGSTDDEV	RUNNINGSUM	RUNNINGVARIANCE
SIGN	SIN	SINH

SPACE	SQRT	STDDEV
SUBSTRING	SUM	TAN
TANH	THIS	VARIANCE

Restrictions for Extended NUMERIC Precision Data Type

The extended NUMERIC precision data type is not supported:

- On the disk for SQL/MP tables
- By Module File caching (MFC)

Example for Extended NUMERIC Precision Data Type

```
>>create table t( n NUMERIC(128,30));

--- SQL operation complete.

>>

>>showddl table t;

CREATE TABLE CAT.SCH.T
  (
    N                          NUMERIC(128, 30) DEFAULT NULL
  )
.....
.....
.....
.....
    ;

--- SQL operation complete.
```

Floating-Point Data

NonStop SQL/MX Release 2.x uses IEEE floating-point format internally and automatically converts Tandem floating-point formats used in host variables or SQL/MP tables. However, that conversion can cause rounding errors, or it can fail for extremely large or extremely small values. Therefore, your programs might experience a difference in the results compared to previous releases of NonStop SQL/MX.

Applications might not be able to retrieve rows using floating-point values in equal comparison predicates by using these columns for some floating-point values.

In addition, if you are using an SQL/MP table that contains FLOAT columns (REAL, DOUBLE PRECISION) with user default values specified, a similarity check for the table with the compile-time default value might fail for some values, and NonStop SQL/MX will recompile the query.

For some queries that use the default value, you might not be able to access a float column with a default value near the boundary value for Tandem float values.

In this example, you are able to perform an ALTER TABLE statement in SQLCI on an SQL/MP table to add a float column with a default value near the boundary value for Tandem float values, but you are unable to use NonStop SQL/MX to insert this default value into the float column. In SQLCI, create an SQL/MP table:

```
>>create table tfloat (c1 INT, c2 INT);
--- SQL operation complete.
>>insert into tfloat (c1) values (10);
--- 1 row(s) inserted.
>>select * from tfloat;

C1           C2
-----------  -----------

         10            ?

--- 1 row(s) selected.
```

You can alter the table to add a float column with the default value 1.15792089237316189e77:

```
>>alter table tfloat add column c3 float(54) default -
1.15792089237316189e77;
--- SQL operation complete.
--
-- Float column c3 has not been populated yet.
--
>>select * from tfloat;

C1           C2           C3
-----------  -----------  ------------------------

         10            ?  -0.11579208923731618E+78
```

But you cannot use NonStop SQL/MX to insert into this table using the default value for the float column:

```
Hewlett-Packard NonStop(TM) SQL/MX Conversational Interface 2.0
(c) Copyright 2003 Hewlett-Packard Development Company, LP.

>> insert into $data16.pnlmx.tfloat(c1,c2) values (12, 13);

*** ERROR[8411] A numeric overflow occurred during an arithmetic
computation or data conversion.

--- 0 row(s) inserted.
```

You can select the rows from the table:

```
>>select * from $vol.subvol.tfloat;
C1 C2 C3
----------- ----------- ------------------------
10 ? -1.15792089237316160E+077
--- 1 row(s) selected.
```

Character String Data Types

Considerations for Character String Data Types
SQL/MP Considerations for Character String Data Types

SQL/MX includes both fixed-length character data and variable-length character data. You cannot compare character data to numeric, datetime, or interval data.

```
character-type is:
     CHAR[ACTER] [(length [CHARACTERS])] [char-set]
       [collate-clause] [UPSHIFT]
   | PIC[TURE] X[(length)] [CHARACTERS] [char-set] [DISPLAY]
       [collate-clause] [UPSHIFT]
   | CHAR[ACTER] VARYING(length) [CHARACTERS][char-set]
       [collate-clause] [UPSHIFT]
   | VARCHAR(length) [CHARACTERS] [char-set]
       [collate-clause] [UPSHIFT]
   | NCHAR [(length)] [CHARACTERS] [collate-clause] [UPSHIFT]
   | NCHAR VARYING (length) [CHARACTERS] [collate-clause]
       [UPSHIFT]
   | NATIONAL CHAR[ACTER] [(length)] [CHARACTERS]
       [collate-clause] [UPSHIFT]
   | NATIONAL CHAR[ACTER] VARYING (length) [CHARACTERS]
      [collate-clause] [UPSHIFT]

char-set is
  CHARACTER SET char-set-name

collate-clause is
  COLLATE collation
```

CHAR, PIC, NCHAR, and NATIONAL CHAR are fixed-length character types. CHAR VARYING, VARCHAR, NCHAR VARYING and NATIONAL CHAR VARYING are varying-length character types.

length

is a positive integer that specifies the number of characters allowed in the column. You must specify a value for *length*.

char-set-name

is the character set name, which can be ISO88591 or UCS2 for any use or KANJI or KSC5601 if the data type is not used to define an SQL/MX column.

collation

is the collation. The only allowed collation is DEFAULT.

The UPSHIFT clause directs NonStop SQL/MX to upshift characters before storing them in the column.

```
CHAR[ACTER] [(length [CHARACTERS])] [char-set] [collate-clause]
   [UPSHIFT]
```

specifies a column with fixed-length character data.

```
PIC[TURE] X[(length)] [DISPLAY] [char-set] [collate-clause]
   [UPSHIFT]
```

specifies a column with fixed-length character data.

You can specify the number of characters in a `PIC X` column by specify either *`length`* or multiple Xs, with each X representing one character position. `DISPLAY` does not change the meaning of the clause.

PIC is an SQL/MX extension.

```
CHAR[ACTER] VARYING (length) [CHARACTERS] [char-set]
   [collate-clause] [UPSHIFT]
```

specifies a column with varying-length character data. `VARYING` specifies that the number of characters stored in the column can be fewer than the *`length`*.

Note that values in a column declared as `VARYING` can be logically and physically shorter than the maximum length, but the maximum internal size of a `VARYING` column is actually four characters larger than the size required for an equivalent column that is not `VARYING`.

```
VARCHAR (length) [char-set] [collate-clause] [UPSHIFT]
```

specifies a column with varying-length character data.

`VARCHAR` is equivalent to data type `CHAR[ACTER] VARYING`.

```
NCHAR [(length)] [collate-clause] [UPSHIFT]
NATIONAL CHAR[ACTER] [(length)] [collate-clause]
       [UPSHIFT]
```

specifies a column with data in the pre-defined national character set.

```
NCHAR VARYING [(length)] [collate-clause] [UPSHIFT]
NATIONAL CHAR[ACTER] VARYING (length) [collate-clause]
   [UPSHIFT]
```

specifies a column with varying-length data in the pre-defined national character set.

Considerations for Character String Data Types

Difference Between CHAR and VARCHAR

You can specify a fixed-length character column as CHAR(*n*), where *n* is the number of characters you want to store. However, if you store five characters into a column specified as CHAR(10), ten characters are stored where the rightmost five characters are blank.

If you do not want to have blanks added to your character string, you can specify a variable-length character column as VARCHAR(*n*), where *n* is the maximum number of characters you want to store. If you store five characters in a column specified as VARCHAR(10), only the five characters are stored logically—without blank padding.

When you are creating SQL/MP tables, group all variable-length columns after all fixed-length columns for faster access. This practice also allows for more efficient use of disk storage. For SQL/MX tables, the executor will put variable-length columns in the most effective place.

Maximum Byte Length of a Character Column

The maximum length of a character column in an SQL/MP Format 1 table depends on whether the data type is fixed-length or variable-length and on the file organization of the file that contains the column. All SQL/MX tables are key-sequenced files.

Data Type	Key-Sequenced	Entry-Sequenced
SQL/MP Format 1 tables:		
Single-byte fixed-length	4061	4072
Single-byte variable-length	4059	4070
Double-byte fixed-length	2030	2036
Double-byte variable-length	2029	2035
SQL/MX tables:		
4K block size	4040*	Not applicable
32K block size	32712**	Not applicable

*The maximum row size is 4040 bytes, but the actual row size is less than that because of bytes used by the header, null indicator, column length indicator, and other system features.

**The maximum row size is 32712 bytes, but the actual row size is less than that because of bytes used by the header, null indicator, column length indicator, and other system features.

For information about the maximum row size available to users, see Table 2-2.

Each variable-length character data item requires eight characters of storage for length information, in addition to the space required for the data itself. As a result, the maximum length for a variable-length column is less than the maximum length for an otherwise equivalent fixed-length column.

A column that allows null value requires two extra storage characters.

Collations and Character Sets

For SQL/MX Release 2.x, a character data type can be associated only with the DEFAULT collation. You set the default NCHAR data type when you install NonStop SQL/MX, and you can select from the ISO88591, UCS2, KANJI or KSC5601 character sets. If you do not make a selection, the default is UCS2.

Note. SQL/MX tables do not support the KANJI or KSC5601 character sets. If you attempt to create an SQL/MX table with these or other unsupported character set types, an SQL error is returned and the operation fails.

For SQL/MP tables, a character data type has an associated character set and collation that can be implicitly or explicitly specified. The CHAR data type can be associated with any of the character sets, and the NCHAR data type is typically associated with the KANJI and KSC5601character sets.

For more information, see Character Sets on page 6-4 and Database Object Names on page 6-13.

NCHAR Columns in SQL/MX and SQL/MP Tables

In NonStop SQL/MX and NonStop SQL/MP, the NCHAR type specification is equivalent to:

- NATIONAL CHARACTER
- NATIONAL CHAR
- CHAR ... CHARACTER SET ..., where the character set is the default character set for NCHAR

Similarly, you can use NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, and VARCHAR ... CHARACTER SET ... .

SQL/MP Considerations for Character String Data Types

Selecting NCHAR Columns

NonStop SQL/MX supports accessing KANJI- or KSC5601-aliased NCHAR columns in an SQL/MP table. For example, suppose that an SQL/MP table has an NCHAR column defined as:

```
MPNcharCol  NCHAR(10)
```

You can select from this column as shown:

```
SELECT MPNcharCol FROM MPTable;
```

This query returns ten characters for each row.

Using the CHAR Keyword in the CAST Operation

You can also use the CAST operation to ensure a certain number of characters are returned from a query. For example:

```
SELECT CAST(MPNcharCol AS CHAR(5) character set KANJI) FROM
   MPTable;
```

This query returns five characters for each row.

You can also insert into or update NCHAR columns in SQL/MP tables. See Inserting Into or Updating SQL/MP NCHAR Columns on page 6-66.

Datetime Data Types

Considerations for Datetime Data Types

[SQL/MP Considerations for Datetime Data Types Not Equivalent to DATE, TIME, TIMESTAMP]
[SQL/MP Considerations for Datetime Data Types Equivalent to DATE, TIME, TIMESTAMP]

A value of datetime data type represents a point in time according to the Gregorian calendar and a 24-hour clock in local civil time (LCT). A datetime item can represent a date, a time, or a date and time.

NonStop SQL/MX accepts dates, such as October 5 to 14, 1582, that were omitted from the Gregorian calendar. This functionality is an SQL/MX extension.

The range of times that a datetime value can represent is:

January 1, 1 A.D., 00:00:00.000000 (low value)
December 31, 9999, 23:59:59.999999 (high value)

NonStop SQL/MX has three datetime data types:

```
datetime-type is:
     DATE
   | TIME [(time-precision)]
   | TIMESTAMP [(timestamp-precision)]
```

`DATE`

specifies a datetime column that contains a date in the external form yyyy-mm-dd and stored in four bytes.

`TIME [(time-precision)]`

specifies a datetime column that, without the optional *time-precision*, contains a time in the external form hh:mm:ss and is stored in three bytes. *time-precision* is an unsigned integer that specifies the number of digits in the fractional seconds and is stored in four bytes. The default for *time-precision* is 0, and the maximum is 6.

`TIMESTAMP [(timestamp-precision)]`

specifies a datetime column that, without the optional *timestamp-precision*, contains a timestamp in the external form yyyy-mm-dd hh:mm:ss and is stored in seven bytes. *timestamp-precision* is an unsigned integer that specifies the number of digits in the fractional seconds and is stored in four bytes. The default for *timestamp-precision* is 6, and the maximum is 6.

Considerations for Datetime Data Types

Datetime Ranges

The range of values for the individual fields in a DATE, TIME, or TIMESTAMP column is specified as:

yyyy	Year, from 0001 to 9999
mm	Month, from 01 to 12
dd	Day, from 01 to 31
hh	Hour, from 00 to 23
mm	Minute, from 00 to 59
ss	Second, from 00 to 59
msssss	Microsecond, from 000000 to 999999

SQL/MP Considerations for Datetime Data Types Not Equivalent to DATE, TIME, TIMESTAMP

When accessing SQL/MP DATETIME columns, you can:

- Select SQL/MP DATETIME columns that are not equivalent to DATE, TIME, or TIMESTAMP.
- Insert into or update SQL/MP DATETIME columns with literals that are not equivalent to DATE, TIME, or TIMESTAMP.

The SQL/MP DATETIME columns that do not map to standard SQL/MX types are represented as SQL/MP DATETIME types in NonStop SQL/MX. These types are:

- DATETIME YEAR
- DATETIME YEAR TO MONTH
- DATETIME YEAR TO HOUR
- DATETIME YEAR TO MINUTE
- DATETIME MONTH
- DATETIME MONTH TO DAY
- DATETIME MONTH TO HOUR
- DATETIME MONTH TO MINUTE
- DATETIME MONTH TO SECOND
- DATETIME MONTH TO FRACTION(n)
- DATETIME DAY
- DATETIME DAY TO HOUR
- DATETIME DAY TO MINUTE
- DATETIME DAY TO SECOND
- DATETIME DAY TO FRACTION(n)
- DATETIME HOUR
- DATETIME HOUR TO MINUTE
- DATETIME HOUR TO SECOND

- DATETIME HOUR TO FRACTION(n)
- DATETIME MINUTE
- DATETIME MINUTE TO SECOND
- DATETIME MINUTE TO FRACTION(n)
- DATETIME SECOND
- DATETIME SECOND TO FRACTION(n)

Selecting DATETIME Columns in SQL/MP Tables

The SQL/MP DATETIME data type has a range of logically contiguous fields in this order: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION.

A specific DATETIME data type consists of a subset or range of these fields and a specified number of significant digits for the FRACTION field. For example:

```
DATETIME YEAR TO MONTH
DATETIME DAY TO FRACTION(3)
```

The qualifier that specifies the range of fields for the DATETIME data type has the same syntax as the qualifier that specifies the range of fields for the INTERVAL data type.

Selecting Supported DATETIME Columns

NonStop SQL/MX supports accessing any SQL/MP DATETIME column—except those consisting of FRACTION only. For example, suppose that an SQL/MP table has a DATETIME column defined as:

```
MPDateTimeCol  DATETIME MONTH TO DAY
                   DEFAULT DATETIME '03-12' MONTH TO DAY
```

You can select from this column as shown:

```
SELECT MPDateTimeCol FROM MPTable;

MPDateTimeCol
-------------
          ...
        03-12
         ...
```

Selecting FRACTION-Only DATETIME Columns

If you attempt to select data from a FRACTION-only DATETIME column, the value is returned as a string of `'#'` characters with the same display length as the length of the column. For example, suppose that an SQL/MP table has a DATETIME column defined as:

```
MPDateTimeCol  DATETIME FRACTION(6)
                   DEFAULT DATETIME '123456' FRACTION(6)
```

You cannot select the data from this column. For example:

```
SELECT MPDateTimeCol FROM MPTable;

MPDateTimeCol
```

```
- - - - - - - - - - - - -
           ######
           ######
              ...
```

NonStop SQL/MX returns a warning indicating that you selected an unsupported data type with undefined contents.

SQL/MP Considerations for Datetime Data Types Equivalent to DATE, TIME, TIMESTAMP

When accessing SQL/MP DATETIME columns, you can:

- Select SQL/MP DATETIME columns that are equivalent to DATE, TIME, or TIMESTAMP.
- Insert into or update SQL/MP DATETIME columns with equivalent DATE, TIME, or TIMESTAMP literals

The SQL/MP DATETIME columns that map to standard SQL/MX types are represented as standard types in NonStop SQL/MX. As a result, the behavior of these SQL/MP data types might be different within NonStop SQL/MX compared to their behavior in NonStop SQL/MP.

The equivalent mappings are:

SQL/MP DATETIME Type	Equivalent SQL/MX Type
DATETIME YEAR TO DAY	DATE
DATETIME YEAR TO SECOND	TIMESTAMP(0)
DATETIME YEAR TO FRACTION(1)	TIMESTAMP(1)
DATETIME YEAR TO FRACTION(2)	TIMESTAMP(2)
DATETIME YEAR TO FRACTION(3)	TIMESTAMP(3)
DATETIME YEAR TO FRACTION(4)	TIMESTAMP(4)
DATETIME YEAR TO FRACTION(5)	TIMESTAMP(5)
DATETIME YEAR TO FRACTION(6)	TIMESTAMP(6) or TIMESTAMP
DATETIME HOUR TO SECOND	TIME(0) or TIME
DATETIME HOUR TO FRACTION(1)	TIME(1)
DATETIME HOUR TO FRACTION(2)	TIME(2)
DATETIME HOUR TO FRACTION(3)	TIME(3)
DATETIME HOUR TO FRACTION(4)	TIME(4)
DATETIME HOUR TO FRACTION(5)	TIME(5)
DATETIME HOUR TO FRACTION(6)	TIME(6)

Using SQL/MX Datetime Functions on DATETIME Data

You can use SQL/MX datetime functions to select individual fields from a DATETIME column in an SQL/MP table. For example, suppose that an SQL/MP table has a DATETIME column defined as:

```
MPDateTimeCol  DATETIME MONTH TO DAY
                   DEFAULT DATETIME '03-12' MONTH TO DAY
```

You can select the month from this column:

```
SELECT MONTH(MPDateTimeCol) FROM MPTable;
```

```
(EXPR)
-------
    ...
      3
   ...
```

See Datetime Functions on page 8-4.

Casting DATETIME Data for Compatibility

DATETIME data types are compatible only if the types have the same start and end fields. No implicit extension or truncation is performed. If the data does not have the same start and end fields, you must use CAST to provide an explicit conversion that allows you to operate on different DATETIME data types.

Overlapping Fields Requirement

You can use CAST provided that the two DATETIME values have at least one overlapping field. This specification is valid because the types overlap on the DAY field:

```
CAST(DATE '2000-03-31' AS DATETIME DAY TO HOUR)
```

However, this specification is not valid because no fields overlap:

```
CAST(DATETIME '2000-03' YEAR TO MONTH AS TIME)
```

Extension Resulting From CAST

Suppose that an SQL/MP table has a DATETIME column defined as:

```
MPDateTimeCol  DATETIME MONTH TO DAY
                   DEFAULT DATETIME '03-12' MONTH TO DAY
```

Use CAST to compare data:

```
SELECT * FROM MPTable
WHERE CAST(MPDateTimeCol AS DATE) > CURRENT_DATE;
```

If extension occurs on the more significant end of a value, the values for the missing fields are drawn from the fields of CURRENT_TIMESTAMP. If extension occurs on the less significant end, the values are the minimum field values. In this example, the YEAR field is from the YEAR field of CURRENT_TIMESTAMP.

Suppose that the current timestamp is 2000-01-26:10:24:10.212072. This expression involves extension on both ends:

```
CAST(DATETIME '12-23' MONTH TO DAY AS TIMESTAMP)
```

The result of the CAST is 2000-12-23:00:00:00.000000.

Operations Equivalent to UNITS

The SQL/MP UNITS operator is not supported. However, NonStop SQL/MX does support equivalent syntax.

Suppose that an SQL/MP table has a DATETIME column defined as:

```
MPDateTimeCol  DATETIME MONTH TO DAY
                   DEFAULT DATETIME '03-12' MONTH TO DAY
```

Using this column as an example, you can specify this equivalent:

SQL/MP Element	SQL/MX Equivalent
`MPDateTimeCol UNITS MONTH`	`MONTH(MPDateTimeCol)`

Interval Data Types

Considerations for Interval Data Types
SQL/MP Considerations for Interval Data Types

Values of interval data type represent durations of time in year-month units (years and months) or in day-time units (days, hours, minutes, seconds, and fractions of a second).

```
interval-type is:
INTERVAL { start-field TO end-field | single-field }

start-field is:
   {YEAR | MONTH | DAY | HOUR | MINUTE} [(leading-precision)]

end-field is:
   YEAR | MONTH | DAY | HOUR | MINUTE | SECOND
        [(fractional-precision)]

single-field is:
   start-field | SECOND [(leading-precision,
                          fractional-precision)]
```

`INTERVAL { start-field TO end-field | single-field }`

specifies a column that represents a duration of time as either a year-month or day-time range or a single-field. The optional sign indicates if this is a positive or negative integer. If you omit the sign, it defaults to positive.

If the interval is specified as a range, the *start-field* and *end-field* must be in one of these categories:

`{YEAR | MONTH | DAY | HOUR | MINUTE} [(`*`leading-precision`*`)]`

specifies the *start-field*. A *start-field* can have a *leading-precision* up to 18 digits (the maximum depends on the number of fields in the interval). The *leading-precision* is the number of digits allowed in the *start-field*. The default for *leading-precision* is 2.

`YEAR | MONTH | DAY | HOUR | MINUTE | SECOND [(`*`fractional-precision`*`)]`

specifies the *end-field*. If the *end-field* is SECOND, it can have a *fractional-precision* up to 6 digits. The *fractional-precision* is the number of digits of precision after the decimal point. The default for *fractional-precision* is 6.

`start-field` `| SECOND [(`*`leading-precision, fractional-precision`*`)]`

specifies the *single-field*. If the *single-field* is SECOND, the *leading-precision* is the number of digits of precision before the decimal point, and the *fractional-precision* is the number of digits of precision after the decimal point.

The default for *leading-precision* is 2, and the default for *fractional-precision* is 6. The maximum for *leading-precision* is 18, and the maximum for *fractional-precision* is 6.

Considerations for Interval Data Types

Interval Leading Precision

The maximum for the *leading-precision* depends on the number of fields in the interval and on the *fractional-precision*. The maximum is computed as:

`max-leading-precision` `= 18 -` *`fractional-precision`* `- 2 * (`*`N`*` - 1)`

where *N* is the number of fields in the interval.

For example, the maximum number of digits for the *leading-precision* in a column with data type INTERVAL YEAR TO MONTH is computed as: 18 – 0 – 2 * (2 – 1) = 16

Interval Ranges

Within the definition of an interval range (other than a single field), the *start-field* and *end-field* can be any of the specified fields with these restrictions:

- An interval range is either year-month or day-time—that is, if the *start-field* is YEAR, the *end-field* is MONTH; if the *start-field* is DAY, HOUR, or MINUTE, the *end-field* is also a time field.
- The *start-field* must precede the *end-field* within the hierarchy: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

Signed Intervals

To include a quoted string in a signed interval data type, the sign must be outside the quoted string. It can be before the entire literal or immediately before the duration enclosed in quotes.

For example, for the interval “minus (5 years 5 months)”, the following formats are valid:

```
INTERVAL - '05-05'YEAR TO MONTH
- INTERVAL '05-05' YEAR TO MONTH
```

Overflow Conditions

When you insert a fractional value into an INTERVAL data type field, if the fractional value is 0 (zero) it does not cause an overflow. Inserting value INTERVAL '1.000000' SECOND(6) into a field SECOND(0) does not cause a loss of value. Provided that the value fits in the target column without a loss of precision, NonStop SQL/MX does not return an overflow error.

However, if the fractional value is > 0, an overflow occurs. Inserting value INTERVAL '1.000001' SECOND(6) causes a loss of value.

SQL/MP Considerations for Interval Data Types

Selecting INTERVAL Columns in SQL/MP Tables

SQL/MP INTERVAL values represent durations of time in year-month units (years and months), in day-time units (days, hours, minutes, seconds, and fractions of a second), or in subsets of those units.

A specific INTERVAL data type consists of a subset or range of these fields and a specified number of significant digits for the FRACTION field if it exists. For example:

```
INTERVAL YEAR TO MONTH
```

```
INTERVAL DAY TO FRACTION(3)
```

Selecting Supported INTERVAL Columns

NonStop SQL/MX supports accessing any INTERVAL column—except those consisting of FRACTION only. For example, suppose that an SQL/MP table has an INTERVAL column defined as:

```
MPIntervalCol  INTERVAL SECOND(2) TO FRACTION(1)
      DEFAULT INTERVAL '36.8' SECOND(2) TO FRACTION(1)
```

You can select from this column as shown:

```
SELECT MPIntervalCol FROM MPTable;
```

```
MPIntervalCol
-------------
          ...
         36.8
          ...
```

In this example, the SQL/MP column of type `INTERVAL SECOND(2) TO FRACTION(1)` is interpreted in NonStop SQL/MX as type `INTERVAL SECOND(2,1)`.

In general, a FRACTION end field in NonStop SQL/MP is interpreted in NonStop SQL/MX as though it were the fractional precision of a SECOND field, provided the start field is SECOND or larger.

Selecting FRACTION-Only INTERVAL Columns

If you attempt to select data from a FRACTION-only INTERVAL column, the column value is returned as a string of `'#'` characters with the same display length as the length of the column.

For example, suppose that an SQL/MP table has an INTERVAL column defined as:

```
MPIntervalCol  INTERVAL FRACTION(6)
                   DEFAULT INTERVAL '123456' FRACTION(6)
```

You can select from this column as shown:

```
SELECT MPIntervalCol FROM MPTable;

MPIntervalCol
-------------
       ######
       ######
         ...
```

NonStop SQL/MX returns a warning indicating that you selected an unsupported data type with undefined contents.

Numeric Data Types

Example for Extended NUMERIC Precision Data Type

Numeric data types are either exact or approximate. A numeric data type is compatible with any other numeric data type, but not with character, datetime, or interval data types.

```
exact-numeric-type is:
     NUMERIC [(precision [,scale])] [SIGNED|UNSIGNED]
   | SMALLINT [SIGNED|UNSIGNED]
   | INT[EGER] [SIGNED|UNSIGNED]
   | LARGEINT
   | DEC[IMAL] [(precision [,scale])] [SIGNED|UNSIGNED]
   | PIC[TURE] [S]{9(integer) [V[9(scale)]] | V9(scale)}
               [DISPLAY [SIGN IS LEADING] | COMP]
approximate-numeric-type is:
     FLOAT [(precision)]
   | REAL
   | DOUBLE PRECISION
```

Exact numeric data types are types that can represent a value exactly: NUMERIC, SMALLINT, INTEGER, LARGEINT, DECIMAL, and PICTURE COMMENT:.

Approximate numeric data types are types that do not necessarily represent a value exactly: FLOAT, REAL, and DOUBLE PRECISION.

A column in an SQL/MP table declared with a floating-point data type is stored in Tandem floating-point format and all computations on it are done assuming that. SQL/MP tables can contain only Tandem floating-point data. For more information about SQL/MP data types, see the *SQL/MP Reference Manual*.

A column in an SQL/MX table declared with a floating-point data type is stored in IEEE floating-point format and all computations on it are done assuming that. SQL/MX tables can contain only IEEE floating-point data. NonStop SQL/MX can select data from both SQL/MP and SQL/MX tables. See default attribute Data Types on page 10-51 for details.

`NUMERIC [(`*`precision`*` [,`*`scale`*`])] [SIGNED|UNSIGNED]`

specifies an exact numeric column, which can be `SIGNED` or `UNSIGNED`.

`precision` specifies the total number of digits and cannot exceed 128.

`scale` specifies the number of digits to the right of the decimal point and cannot exceed *`precision`*.

For signed numbers with a precision up to 9 and unsigned numbers with a precision of up to 18, the number is stored internally in binary and is supported in hardware. In all other cases, the number is supported in software, which is less efficient.

The default is `NUMERIC (9,0) SIGNED`.

`SMALLINT [SIGNED|UNSIGNED]`

specifies an exact numeric column—a two-byte binary integer, `SIGNED` or `UNSIGNED`. The column stores integers in the range unsigned 0 to 65535 or signed -32768 to +32767.

The default is `SIGNED`.

`INT[EGER] [SIGNED|UNSIGNED]`

specifies an exact numeric column—a four-byte binary integer, `SIGNED` or `UNSIGNED`. The column stores integers in the range unsigned 0 to 4294967295 or signed -2147483648 to +2147483647.

The default is `SIGNED`.

`LARGEINT`

specifies an exact numeric column—an eight-byte signed binary integer. The column stores integers in the range -2**63 to 2**63 -1 (approximately 9.223 times 10 to the eighteenth power).

`DEC[IMAL] [(`*`precision`*` [,`*`scale`*`])] [SIGNED|UNSIGNED]`

specifies an exact numeric column—a decimal number, `SIGNED` or `UNSIGNED`, stored as ASCII characters. *`precision`* specifies the total number of digits and cannot exceed 18. If *`precision`* is 10 or more, the value must be `SIGNED`. The sign is stored as the first bit of the leftmost byte. *`scale`* specifies the number of digits to the right of the decimal point.

The default is `DECIMAL (9,0) SIGNED`.

```
PIC[TURE] [S]{ 9(integer) [V[9(scale)]] | V9(scale) }
          [DISPLAY [SIGN IS LEADING] | COMP]
```

specifies an exact numeric column. If you specify `COMP`, the column is binary and equivalent to the data type `NUMERIC`. If you omit `COMP`, `DISPLAY [SIGN IS LEADING]` is the default, and the data type is equivalent to the data type `DECIMAL`.

The S specifies a signed column. The sign is stored as the first bit of the leftmost byte (digit). If you omit S, the column is unsigned. A column with ten or more digits must be signed.

The `9(`*`integer`*`)` specifies the number of digits in the integral part of the value. The V designates a decimal position. The `9(`*`scale`*`)` designates the number of positions to the right of the decimal point. If you omit `V9` (*`scale`*), the scale is 0. If you specify only `V9`, the scale is 1.

Instead of *`integer`* or *`scale`*, you can specify multiple 9s, with each 9 representing one digit. For example, `PIC 9V999` has a scale of 3. The values of *`integer`* and *`scale`* determine the length of the column. The sum of these values cannot exceed 18.

There is no default. You must specify either `9(`*`integer`*`)` or `V9` (*`scale`*).

`FLOAT [(`*`precision`*`)]`

specifies an approximate numeric column. The column stores floating-point numbers and designates from 1 through 52 bits of *`precision`*. The range is from +/- 2.2250738585072014e-308 through +/-1.7976931348623157e+308 stored in 8 bytes.

An IEEE FLOAT *`precision`* data type is stored as an IEEE DOUBLE, that is, in 8 bytes, with the specified precision.

The default *`precision`* is 52.

`REAL`

specifies a 4-byte approximate numeric column. The column stores 32-bit floating-point numbers with 23 bits of binary precision and 8 bits of exponent.

The minimum and maximum range is from +/- 1.17549435e-38 through +/ 3.40282347e+38.

`DOUBLE PRECISION`

specifies an 8-byte approximate numeric column.

The column stores 64-bit floating-point numbers and designates from 1 through 52 bits of *`precision`*.

An IEEE DOUBLE PRECISION data type is stored in 8 bytes with 52 bits of binary precision and 11 bits of exponent. The minimum and maximum range is from +/- 2.2250738585072014e-308 through +/-1.7976931348623157e+308.

DEFINEs

A DEFINE is a named set of attribute-value pairs associated with a process. You can use DEFINEs to pass information to a process when you start the process. DEFINEs are often used to pass information about Guardian names. DEFINEs can be used only for SQL/MP objects.

NonStop SQL/MX allows you to use DEFINE names as logical names for tables, views, or partitions in SQL/MX statements that query SQL/MP objects. When NonStop SQL/MX compiles such statements, it replaces the DEFINE name in the statement with the Guardian name currently associated with the DEFINE.

A DEFINE name begins with an equal sign (=) followed by a letter and can contain 1 to 24 characters, including alphanumeric characters and underscores (_). Uppercase and lowercase characters are considered equivalent in DEFINE names.

A DEFINE name must not be a reserved word. Otherwise, you cannot select data using the DEFINE name of the table, view, or partition. See Appendix B, Reserved Words.

The reasons for using DEFINE names in SQL/MX statements are as follows:

- DEFINE names are easier to understand than Guardian names.

  For example, the name =CUSTOMER is simpler than an actual file name such as \MYSYS.$SAMDB.SALES.CUSTOMER.

- DEFINE names provide location independence.

  For example, if you code with DEFINE names, you can rename database objects, move database objects, or change the database that a program accesses without changing source code.

Using DEFINEs

DEFMODE is an attribute of a process that controls whether you can create DEFINEs from the process and whether DEFINEs are propagated when the process starts another process. The process can be a TACL process, an OSS shell process, an MXCI process, or a process of your own creation. The DEFMODE attribute is ON by default but can be set to OFF in TACL or the OSS shell.

When DEFMODE is ON, you can create, modify, delete, propagate, and display information about DEFINEs. For example, if you start an MXCI process from a TACL process with DEFMODE ON, DEFINEs set in the TACL process are propagated to the MXCI process. Similarly, you can set DEFINEs in an OSS shell process and the DEFINEs are propagated to a process you start from an OSS program with embedded SQL statements. DEFMODE ON is the default. Note that for OSS processes, DEFMODE ON becomes the default after the first add_define command is issued.

When DEFMODE is OFF, DEFINEs are ignored, and you cannot create new DEFINEs. You can still modify, delete, and display information about existing DEFINEs, but such DEFINEs have no effect because they are not propagated to other programs.

Use these commands to work with DEFINEs from MXCI. Each command is described in more detail in a separate entry.

ADD DEFINE Command on page 4-4	Adds a DEFINE in the current MXCI session
ALTER DEFINE Command on page 4-6	Changes the physical name of a DEFINE in the current MXCI session
DELETE DEFINE Command on page 4-9	Deletes a DEFINE in the current MXCI session
INFO DEFINE Command on page 4-45	Displays the logical and physical names of DEFINEs in the current MXCI session

TACL has similar commands with the same names as the MXCI commands just listed. The OSS shell has similar commands, add_define, del_define, info_define, set_define, and show_define. See the *TACL Reference Manual* or the *Open System Services Shell and Utilities Reference Manual* for more information about DEFINE-related commands in TACL or the OSS shell, respectively.

Use these system procedures to work with DEFINEs from within an SQL program. See the *Guardian Procedure Calls Reference Manual* or the *Open System Services System Calls Reference Manual* for more information about the procedures.

DEFINEADD	Adds a DEFINE
CHECKDEFINE	Checkpoints a DEFINE to a backup process
DEFINEDELETE	Deletes DEFINEs
DEFINEDELETEALL	Deletes all DEFINEs except =_DEFAULTS from the context of the current process
DEFINEINFO	Returns DEFINE attribute values
DEFINEMODE	Enables or disables the use of DEFINEs
DEFINEEXTNAME	Returns the next DEFINE name (DEFINEs are stored in ascending order by name)
DEFINEREADATTR	Returns an attribute value for a DEFINE or for the working attribute set
DEFINERESTOREWORK	Restores the working attribute set from the background set
DEFINESAVEWORK	Saves the working attribute set in the background set
DEFINESETATTR	Alters the value of an attribute in the working set, or resets the attribute
DEFINESETLIKE	Sets all attributes of the working set to match those of an existing DEFINE
DEFINEVALIDATEWORK	Checks the working set for consistency and completeness

Using DEFINEs From MXCI

- Make sure DEFMODE is set to ON in TACL or the OSS shell.

  DEFMODE is ON by default but can be set to OFF in TACL or the OSS shell. To inherit DEFINEs from the process that starts MXCI, such as TACL or the OSS shell, verify that DEFMODE is ON before you start MXCI. If you set DEFMODE OFF before you start MXCI, you will not inherit DEFINEs, and you will not be able to create new DEFINEs during the MXCI session.

  For more information on how to show or change the DEFMODE setting in TACL or the OSS shell, see the *TACL Reference Manual* or the *Open System Services Shell and Utilities Reference Manual*, respectively.

- DEFINEs that you create during an MXCI session remain in effect until you alter them, delete them, or end the MXCI session. DEFINEs you inherit from another process and then modify with MXCI commands revert to their previous attribute values (that is, the values they had when you started MXCI) when you end the MXCI session. Any changes you make to inherited attributes within the MXCI session apply only until you exit MXCI.

- MXCI resolves DEFINE names in a statement at the time you enter or execute the statement.

For more information on using DEFINEs with SQL programs, see the *SQL/MX Programming Manual for C and COBOL*.

DEFINEs of Class MAP

In NonStop SQL/MX, DEFINEs can have only one CLASS attribute, the MAP class. A DEFINE of class MAP associates a DEFINE name with the name of a table, view, or partition. You can use the DEFINE name in SQL statements as the logical name of a table, view, or partition, altering the DEFINE (but not the SQL statement) when you want to point to a different physical entity.

For example, this command adds a DEFINE that assigns the logical name =ORDERS to the table whose name is $SAMDB.SALES.ORDERS:

```
ADD DEFINE =ORDERS, CLASS MAP, FILE $SAMDB.SALES.ORDERS;
```

While this DEFINE is in effect, you can refer to the table as =ORDERS in SQL statements.

MAP is the default class, so the previous command is normally equivalent to:

```
ADD DEFINE =ORDERS, FILE $SAMDB.SALES.ORDERS;
```

Expressions

An SQL value expression, referred to as an expression, can evaluate to a value with one of these:

Character Value Expressions on page 6-41	Operands can be combined with the concatenation operator (||). Example: `'HOUSTON,' ||' TEXAS'`
Datetime Value Expressions on page 6-43	Operands can be combined in specific ways with arithmetic operators. Example: `CURRENT_DATE + INTERVAL '1' DAY`
Interval Value Expressions on page 6-47	Operands can be combined in specific ways with addition and subtraction operators. Example: `INTERVAL '2' YEAR - INTERVAL '3' MONTH`
Numeric Value Expressions on page 6-52	Operands can be combined in specific ways with arithmetic operators. Example: `SALARY * 1.10`
Rowset Expressions on page 6-55	Operands can be combined to form rowset expressions.

The data type of an expression is the data type of the value of the expression.

A value expression can be, among other things, a character string literal, a numeric literal, a host variable, a dynamic parameter, or a column name that specifies the value of the column in a row of a table. A value expression can also include, among other operands, functions and scalar subqueries.

Character Value Expressions

The operands of a character value expression—referred to as character primaries—can be combined with the concatenation operator (||). The data type of a character primary is character string.

```
character-expression is:
     character-primary
   | character-expression || character-primary

character-primary is:
     character-string-literal
   | column-reference
   | character-type-host-variable
   | dynamic parameter
   | character-value-function
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (character-expression)
```

Character (or string) value expressions are built from operands that can be:

- Character string literals
- Character string functions
- Column references with character values
- Host variables of type CHAR, VARCHAR, and PIC X(l)
- Dynamic parameters
- CURRENT_USER, SESSION_USER, and USER functions
- Aggregate functions, sequence functions, scalar subqueries, CASE expressions, or CAST expressions that return character values

Examples of Character Value Expressions

These are examples of character value expressions:

Expression	Description
`'ABILENE'`	Character string literal.
`'ABILENE ' ||' TEXAS'`	The concatenation of two string literals.
`'ABILENE ' ||' TEXAS' || x’55 53 41`	The concatenation of three string literals to form the literal: 'ABILENE TEXAS USA'
`'Customer ' || custname`	The concatenation of a string literal with the value in column CUSTNAME.
`CAST (order_date AS CHAR)`	CAST function applied to a DATE value.

Datetime Value Expressions

[SQL/MP Considerations for Datetime Value Expressions](#)
[Considerations for Datetime Value Expressions](#)
[Examples of Datetime Value Expressions](#)

The operands of a datetime value expression can be combined in specific ways with arithmetic operators.

In this syntax diagram, the data type of a datetime primary is DATE, TIME, or TIMESTAMP. The data type of an interval term is INTERVAL.

```
datetime-expression is:
     datetime-primary
   | interval-expression + datetime-primary
   | datetime-expression + interval-term
   | datetime-expression - interval-term

datetime-primary is:
     datetime-literal
   | column-reference
   | datetime-type-host-variable
   | dynamic parameter
   | datetime-value-function
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (datetime-expression)

interval-term is:
     interval-factor
   | numeric-term  * interval-factor

interval-factor is:
   [+|-] interval-primary

interval-primary is:
     interval-literal
   | column-reference
   | interval-type-host-variable
   | dynamic parameter
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (interval-expression)
```

Datetime value expressions are built from operands that can be:

- Interval value expressions
- Datetime or interval literals
- Host variables of type DATE, TIME, TIMESTAMP, and INTERVAL
- Dynamic parameters
- Column references with datetime or interval values
- Host variables of type INTERVAL
- Dynamic parameters
- Datetime or interval value functions
- Any aggregate functions, sequence functions, scalar subqueries, CASE expressions, or CAST expressions that return datetime or interval values

SQL/MP Considerations for Datetime Value Expressions

FRACTION-Only DATETIME Columns

Suppose that an SQL/MP table has a DATETIME column defined as:

```
MPDateTimeCol  DATETIME FRACTION(6)
                    DEFAULT DATETIME '123456' FRACTION(6)
```

You cannot use this column in a datetime expression, as a CAST argument, or as an argument of an aggregate function such as MIN or MAX. NonStop SQL/MX returns an error indicating that operations with FRACTION-only columns are not supported.

Considerations for Datetime Value Expressions

Data Type of Result

In general, the data type of the result is the data type of the *datetime-primary* part of the datetime expression. For example, datetime value expressions include:

`CURRENT_DATE + INTERVAL '1' DAY`	The sum of the current date and an interval value of one day.
`CURRENT_DATE + est_complete`	The sum of the current date and the interval value in column EST_COMPLETE.
`( SELECT ship_timestamp FROM project WHERE projcode=1000 ) + INTERVAL '07:04' DAY TO HOUR`	The sum of the ship timestamp for the specified project and an interval value of seven days, four hours.

The datetime primary in the first expression is CURRENT_DATE, a function that returns a value with DATE data type. Therefore, the data type of the result is DATE.

In the last expression, the datetime primary is this scalar subquery:

```
( SELECT ship_timestamp FROM project WHERE projcode=1000 )
```

The preceding subquery returns a value with TIMESTAMP data type. Therefore, the data type of the result is TIMESTAMP.

Restrictions on Operations With Datetime or Interval Operands

You can use datetime and interval operands with arithmetic operators in a datetime value expression only in these combinations:

Operand 1	Operator	Operand 2	Result Type
Datetime	+ or –	Interval	Datetime
Interval	+	Datetime	Datetime

When using these operations, note:

- Adding or subtracting an interval of months to a DATE value results in a value of the same day plus or minus the specified number of months. Because different months have different lengths, this is an approximate result.
- Datetime and interval arithmetic can yield unexpected results, depending on how the fields are used. For example, execution of this expression (evaluated left to right) returns an error:

  ```
  DATE '1996-01-30' + INTERVAL '1' MONTH + INTERVAL '7' DAY
  ```

  In contrast, this expression (which adds the same values as the previous expression, but in a different order) correctly generates the value 1996-03-06:

  ```
  DATE '1996-01-30' + INTERVAL '7' DAY + INTERVAL '1' MONTH
  ```

Examples of Datetime Value Expressions

The PROJECT table consists of five columns that use the data types NUMERIC, VARCHAR, DATE, TIMESTAMP, and INTERVAL DAY. Suppose that you have inserted values into the PROJECT table. For example:

```
INSERT INTO persnl.project
VALUES (1000,'SALT LAKE CITY',DATE '1996-04-10',
       TIMESTAMP '1996-04-21:08:15:00.00',INTERVAL '15' DAY);
```

The next examples use these values in the PROJECT table:

PROJCODE	START_DATE	SHIP_TIMESTAMP	EST_COMPLETE
1000	1996-04-10	1996-04-21 08:15:00.00	15
945	1996-10-20	1996-12-21 08:15:00.00	30
920	1996-02-21	1996-03-12 09:45:00.00	20
134	1996-11-20	1997-01-01 00:00:00.00	30

- Add an interval value qualified by YEAR to a datetime value:

```
SELECT start_date + INTERVAL '1' YEAR
FROM persnl.project
WHERE projcode = 1000;

(EXPR)
----------
1997-04-10

--- 1 row(s) selected.
```

- Subtract an interval value qualified by MONTH from a datetime value:

```
SELECT ship_timestamp - INTERVAL '1' MONTH
FROM persnl.project
WHERE projcode = 134;

(EXPR)
--------------------------
1996-12-01 00:00:00.000000

--- 1 row(s) selected.
```

  The result is 1996-12-01 00:00:00.00. The YEAR value is decremented by 1 because subtracting a month from January 1 causes the date to be in the previous year.

- Add a column whose value is an interval qualified by DAY to a datetime value:

```
SELECT start_date + est_complete
FROM persnl.project
WHERE projcode = 920;

(EXPR)
----------
1996-03-12

--- 1 row(s) selected.
```

  The result of adding 20 days to 1996-02-21 is 1996-03-12. NonStop SQL/MX correctly handles 1996 as a leap year.

- Subtract an interval value qualified by HOUR TO MINUTE from a datetime value:

```
SELECT ship_timestamp - INTERVAL '15:30' HOUR TO MINUTE
FROM persnl.project
WHERE projcode = 1000;

(EXPR)
--------------------------
1996-04-20 16:45:00.000000
```

  The result of subtracting 15 hours and 30 minutes from 1996-04-21 08:15:00.00 is 1996-04-20 16:45:00.00.

Interval Value Expressions

[SQL/MP Considerations for Interval Value Expressions](#)
[Considerations for Interval Value Expressions](#)
[Examples of Interval Value Expressions](#)

The operands of an interval value expression can be combined in specific ways with addition and subtraction operators. In this syntax diagram, the data type of a datetime expression is DATE, TIME, or TIMESTAMP; the data type of an interval term or expression is INTERVAL.

```
interval-expression is:
     interval-term
   | interval-expression + interval-term
   | interval-expression - interval-term
   | (datetime-expression - datetime-primary)
        [interval-qualifier]

interval-term is:
     interval-factor
   | interval-term * numeric-factor
   | interval-term / numeric-factor
   | numeric-term  * interval-factor

interval-factor is:
   [+|-] interval-primary

interval-primary is:
     interval-literal
   | column-reference
   | interval-type-host-variable
   | dynamic parameter
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (interval-expression)

numeric-factor is:
     [+|-] numeric-primary
   | [+|-] numeric-primary ** numeric-factor
```

```
numeric-primary is:
     unsigned-numeric-literal
   | column-reference
   | numeric-type-host-variable
   | dynamic parameter
   | numeric-value-function
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (numeric-expression)

interval-qualifier is:
   start-field TO end-field | single-field

start-field is:
   {YEAR | MONTH | DAY | HOUR | MINUTE} [(leading-precision)]

end-field is:
   YEAR | MONTH | DAY | HOUR | MINUTE | SECOND [(fractional-
                                              precision)]

single-field is:
   start-field | SECOND [(leading-precision,
                           fractional-precision)]
```

Interval value expressions are built from operands that can be:

- Integers
- Datetime value expressions
- Interval literals
- Column references with datetime or interval values
- Host variables of type INTERVAL
- Dynamic parameters
- Datetime or interval value functions
- Aggregate functions, sequence functions, scalar subqueries, CASE expressions, or CAST expressions that return interval values

For `interval-term`, `datetime-expression`, and `datetime-primary`, see Datetime Value Expressions on page 6-43.

If the interval expression is the difference of two datetime expressions, by default, the result is expressed in the least significant unit of measure for that interval. For date differences, the interval is expressed in days. For timestamp differences, the interval is expressed in fractional seconds.

If the interval expression is the difference or sum of interval operands, the interval qualifiers of the operands are either year-month or day-time. If you are updating or inserting a value that is the result of adding or subtracting two interval qualifiers, the interval qualifier of the result depends on the interval qualifier of the target column.

SQL/MP Considerations for Interval Value Expressions

FRACTION-Only Interval Columns

Suppose that an SQL/MP table has an INTERVAL column defined as:

```
MPDateTimeCol  INTERVAL FRACTION(6)
                    DEFAULT INTERVAL '123456' FRACTION(6)
```

You cannot use this column in an interval expression, as a CAST argument, or as an argument of an aggregate function such as MIN or MAX. NonStop SQL/MX returns an error indicating that operations with FRACTION-only columns are not supported.

Considerations for Interval Value Expressions

Start and End Fields

Within the definition of an interval range, the *start-field* and *end-field* can be any of the specified fields with these restrictions:

- An interval is either year-month or day-time. If the *start-field* is YEAR, the *end-field* is MONTH; if the *start-field* is DAY, HOUR, or MINUTE, the *end-field* is also a time field.
- The *start-field* must precede the *end-field* within the hierarchy YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

Within the definition of an interval expression, the *start-field* and *end-field* of all operands in the expression must be either year-month or day-time.

Interval Qualifier

The rules for specifying the interval qualifier of the result expression vary. For example, interval value expressions include:

`CURRENT_DATE - start_date`	By default, the interval difference between the current date and the value in column START_DATE is expressed in days. You are not required to specify the interval qualifier.
`INTERVAL '3' DAY - INTERVAL '2' DAY`	The difference of two interval literals. The result is 1 day.
`INTERVAL '3' DAY + INTERVAL '2' DAY`	The sum of two interval literals. The result is 5 days.
`INTERVAL '2' YEAR - INTERVAL '3' MONTH`	The difference of two interval literals. The result is 1 year, 9 months.

Restrictions on Operations

You can use datetime and interval operands with arithmetic operators in an interval value expression only in these combinations:

Operand 1	Operator	Operand 2	Result Type
Datetime	–	Datetime	Interval
Interval	+ or –	Interval	Interval
Interval	* or /	Numeric	Interval
Numeric	*	Interval	Interval

This table lists valid combinations of datetime and interval arithmetic operators, and the data type of the result:

Operands	Result type
Date + Interval or Interval + Date	Date
Date – Interval	Date
Date – Date	Interval
Time + Interval or Interval + Time	Time
Time – Interval	Time
Timestamp + Interval or Interval + Timestamp	Timestamp
Timestamp – Interval	Timestamp
year-month Interval + year-month Interval	year-month Interval
day-time Interval + day-time Interval	day-time Interval
year-month Interval – year-month Interval	year-month Interval
day-time Interval – day-time Interval	day-time Interval
Time – Time	Interval
Timestamp – Timestamp	Interval
Interval * Number or Number * Interval	Interval
Interval / Number	Interval
Interval – Interval or Interval + Interval	Interval

When using these operations, note:

- If you subtract a datetime value from another datetime value, both values must have the same data type. To get this result, use the CAST expression. For example:

  ```
  CAST (ship_timestamp AS DATE) - start_date
  ```

- If you subtract a datetime value from another datetime value, and you specify the interval qualifier, you must allow for the maximum number of digits in the result for the precision. For example:

  ```
  (CURRENT_TIMESTAMP - ship_timestamp) DAY(4) TO SECOND(6)
  ```

- If you are updating a value that is the result of adding or subtracting two interval values, an SQL error occurs if the source value does not fit into the target column's range of interval fields. For example, this expression cannot replace an INTERVAL DAY column:

  ```
  INTERVAL '1' MONTH + INTERVAL '7' DAY
  ```

- If you multiply or divide an interval value by a numeric value expression, NonStop SQL/MX converts the interval value to its least significant subfield and then multiplies or divides it by the numeric value expression. The result has the same fields as the interval that was multiplied or divided. For example, this expression returns the value `5-02`:

  ```
  INTERVAL '2-7' YEAR TO MONTH * 2
  ```

Examples of Interval Value Expressions

The PROJECT table consists of six columns using the data types NUMERIC, VARCHAR, DATE, TIMESTAMP, and INTERVAL DAY. Suppose that you have inserted values into the PROJECT table. For example:

```
INSERT INTO persnl.project
VALUES (1000,9657,'SALT LAKE CITY',DATE '1996-04-10',
        TIMESTAMP '1996-04-21:08:15:00.00',INTERVAL '15' DAY);
```

The next example uses these values in the PROJECT table:

PROJCODE	START_DATE	SHIP_TIMESTAMP	EST_COMPLETE
1000	1996-04-10	1996-04-21:08:15:00.0000	15
2000	1996-06-10	1996-07-21:08:30:00.0000	30
2500	1996-10-10	1996-12-21:09:00:00.0000	60
3000	1996-08-21	1996-10-21:08:10:00.0000	60
4000	1996-09-21	1996-10-21:10:15:00.0000	30
5000	1996-09-28	1996-10-28:09:25:01.1111	30

- Suppose that the CURRENT_TIMESTAMP is 2000-01-06 11:14:41.748703. Find the number of days, hours, minutes, seconds, and fractional seconds in the difference of the current timestamp and the SHIP_TIMESTAMP in the PROJECT table:

  ```
  SELECT projcode,
    (CURRENT_TIMESTAMP - ship_timestamp) DAY(4) TO SECOND(6)
  FROM samdbcat.persnl.project;

  Project/Code  (EXPR)
  ```

```
------------  ---------------------
        1000   1355 02:58:57.087086
        2000   1264 02:43:57.087086
        2500   1111 02:13:57.087086
        3000   1172 03:03:57.087086
        4000   1172 00:58:57.087086
        5000   1165 01:48:55.975986

--- 6 row(s) selected.
```

Numeric Value Expressions

Considerations for Numeric Value Expressions
Examples of Numeric Value Expressions

The operands of a numeric value expression can be combined in specific ways with arithmetic operators. In this syntax diagram, the data type of a term, factor, or numeric primary is numeric.

```
numeric-expression is:
     numeric-term
   | numeric-expression + numeric-term
   | numeric-expression - numeric-term

numeric-term is:
     numeric-factor
   | numeric-term * numeric-factor
   | numeric-term / numeric-factor

numeric-factor is:
     [+|-] numeric-primary
   | [+|-] numeric-primary ** numeric-factor

numeric-primary is:
     unsigned-numeric-literal
   | column-reference
   | numeric-type-host-variable
   | dynamic parameter
   | numeric-value-function
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (numeric-expression)
```

As shown in the preceding syntax diagram, numeric value expressions are built from operands that can be:

- Numeric literals
- Column references with numeric values

- Host variables of type NUMERIC, PIC S9()V9(), DECIMAL, SMALLINT, INTEGER, LARGEINT, FLOAT, REAL, and DOUBLE PRECISION
- Dynamic parameters
- Numeric value functions
- Aggregate functions, sequence functions, scalar subqueries, CASE expressions, or CAST expressions that return numeric values

Considerations for Numeric Value Expressions

Order of Evaluation

1. Expressions within parentheses
2. Unary operators
3. Exponentiation
4. Multiplication and division
5. Addition and subtraction

Operators at the same level are evaluated from left to right for all operators except exponentiation. Exponentiation operators at the same level are evaluated from right to left. For example, X + Y + Z is evaluated as (X + Y) + Z, whereas X ** Y ** Z is evaluated as X ** (Y ** Z).

Additional Rules for Arithmetic Operations

Numeric expressions are evaluated according to these additional rules:

- An expression with a numeric operator evaluates to null if any of the operands is null.
- Dividing by 0 causes an error.
- Exponentiation is allowed only with numeric data types. If the first operand is 0 (zero), the second operand must be greater than 0, and the result is 0. If the second operand is 0, the first operand cannot be 0, and the result is 1. If the first operand is negative, the second operand must be a value with an exact numeric data type and a scale of zero.
- Exponentiation is subject to rounding error. In general, results of exponentiation should be considered approximate.

Precision, Magnitude, and Scale of Arithmetic Results

The precision, magnitude, and scale are computed during the evaluation of an arithmetic expression. Precision is the maximum number of digits in the expression. Magnitude is the number of digits to the left of the decimal point. Scale is the number of digits to the right of the decimal point.

For example, a column declared as NUMERIC (18, 5) has a precision of 18, a magnitude of 13, and a scale of 5. As another example, the literal 12345.6789 has a precision of 9, a magnitude of 5, and a scale of 4.

The maximum precision for exact numeric data types is 18 digits. The maximum precision for the REAL data type is approximately 7 decimal digits, and the maximum precision for the DOUBLE PRECISION data type is approximately 16 digits.

When NonStop SQL/MX encounters an arithmetic operator in an expression, it applies these rules (with the restriction that if the precision becomes greater than 18, the resulting precision is set to 18 and the resulting scale is the maximum of 0 and (18-(`resulted precision` - `resulted scale`)).

If the operator is + or -, the resulting scale is the maximum of the scales of the operands. The resulting precision is the maximum of the magnitudes of the operands, plus the scale of the result, plus 1.

- If the operator is *, the resulting scale is the sum of the scales of the operands. The resulting precision is the sum of the magnitudes of the operands and the scale of the result.
- If the operator is /, the resulting scale is the sum of the scale of the numerator and the magnitude of the denominator. The resulting magnitude is the sum of the magnitude of the numerator and the scale of the denominator.

For example, if the numerator is NUMERIC (7, 3) and the denominator is NUMERIC (7, 5), the resulting scale is 3 plus 2 (or 5), and the resulting magnitude is 4 plus 5 (or 9). The expression result is NUMERIC (14, 5).

Conversion of Numeric Types for Arithmetic Operations

NonStop SQL/MX automatically converts between floating-point numeric types (REAL and DOUBLE PRECISION) and other numeric types. All numeric values in the expression are first converted to binary, with the maximum precision needed anywhere in the evaluation. The maximum precision for exact numeric data types is 18 digits. The maximum precision for REAL and DOUBLE PRECISION data types is approximately 16.5 digits (54 bits).

NonStop SQL/MX converts floating-point data types following these rules:

- NonStop SQL/MX cannot convert a Tandem REAL or a FLOAT data type with precision between 1 and 22 bits to IEEE REAL, because the Tandem exponent will not fit in an IEEE REAL data type. The precision of a Tandem data type will be maintained correctly.
- There is no equivalent to a Tandem REAL in IEEE floating-point data type which preserves the precision and exponent. If you want a small floating-point data type with less exponent and less storage, declare columns or host variables as REAL. If you want more exponent and more precision, declare it as DOUBLE or FLOAT.

  Suppose that you have an SQL/MP table that includes a column, *mympcol*, declared as REAL. If you create an SQL/MX table with a column *mymxcol*, declared as REAL, you would not be able to convert the SQL/MP column *mympcol* into the SQL/MX column *mymxcol*. You should declare the SQL/MX column as type FLOAT or DOUBLE PRECISION.

Examples of Numeric Value Expressions

These are examples of numeric value expressions:

`-57`	Numeric literal.
`salary * 1.10`	The product of the values in the SALARY column and a numeric literal.
`unit_price * qty_ordered`	The product of the values in the UNIT_PRICE and QTY_ORDERED columns.
`12 * (7 - 4)`	An expression whose operands are numeric literals.
`COUNT (DISTINCT city)`	Function applied to the values in a column.

Rowset Expressions

An expression that contains a rowset host variable or rowset parameter as one of its operands is called a *rowset expression*. A rowset expression is an array of single value expressions, where the operands for the *n*th single value expression are obtained from the *n*th rowset element. All array elements in a given rowset expression are of identical type.

For more information about rowsets, see the *SQL/MX Programming Manual for C and COBOL*.

Identifiers

SQL/MP Considerations for Identifiers
Examples of Identifiers

SQL identifiers are names used to identify tables, views, columns, and other SQL entities. The two types of identifiers are regular and delimited. A delimited identifier is enclosed in double quotes ("). An identifier of either type can contain up to 128 characters.

Regular Identifiers

Regular identifiers begin with a letter (A through Z or a through z), but can also contain digits (0 through 9), or underscore characters (_). Regular identifiers are not case-sensitive. You cannot use a reserved word as a regular identifier.

Delimited Identifiers

Delimited identifiers are character strings that appear within double quote characters (") and consist of alphanumeric characters and other characters except for the at sign (@), the forward slash (/), backward slash (\), and circumflex (^). To include a double quote character in a delimited identifier, use two consecutive double quotes (for example, `"da Vinci's ""Mona Lisa"""`).

Unlike regular identifiers, delimited identifiers are case-sensitive. Spaces within a delimited identifier are significant except for trailing spaces, which NonStop SQL/MX truncates. You can use reserved words as delimited identifiers.

These forms of delimited identifiers are not supported. Results are unpredictable for delimited identifiers that:

- Start with a “\” or “$”
- Consist of space characters only (for example, " ", "   ")
- Consist of special characters only (for example, "~" or "~!#$%^&")
- Contain more than two consecutive double quote characters (for example, """""""""""""""abc""""""""""""""")
- Contain dots (for example, "cat.sch".sch2."cat3.sch3.mod")
- Cause a length limit (128) overflow (for example, 250 double quotes will result in character length of 125 bytes)

Specifying Delimited Identifiers in OSS Command-Line Arguments

Occasionally, you might want to use SQL reserved words such as TIME and ZONE as identifiers to name some of your SQL objects. SQL provides delimited identifiers specifically for these situations. Suppose you have chosen the name TIME for one of your catalogs and the name ZONE for a schema within that catalog. You can pass these delimited identifier names as command-line arguments to an OSS hosted preprocessor invocation by using an escape character for their quotes.

```
mxsqlco prog.cob -g moduleSchema="\"TIME\".\"ZONE\""
```

Suppose that prog.cob has this module directive:

```
EXEC SQL MODULE progmod END-EXEC.
```

The preprocessor invocation preprocesses this module directive as if it were:

```
EXEC SQL MODULE "TIME"."ZONE".progmod END-EXEC.
```

SQL/MP Considerations for Identifiers

Using SQL/MX Reserved Words in SQL/MP Names

Do not use reserved words as identifiers. See Appendix B, Reserved Words.

If an SQL/MP object or column name contains SQL/MX reserved words, you must delimit that part of the Guardian name, either in the SQL/MX statement or in the CREATE SQLMP ALIAS statement, by enclosing the reserved word in double quotes.

For example, suppose that a table has the Guardian name, `ALLOCATE.DESCRIBE`. In NonStop SQL/MX, you must enclose both parts of the name in double quotes because both parts of the name are reserved words. The delimited name is `"ALLOCATE"."DESCRIBE."` If either part of the name is a reserved word, enclose only the part that is a reserved word in double quotes.

When you delimit a column name in NonStop SQL/MX, the column name must be in uppercase letters, because NonStop SQL/MP stores the identifier of the column (or of any SQL entity that is not a physical object) in uppercase. The Guardian names of SQL/MP tables, views, and other physical objects are case insensitive and are not required to be in uppercase letters when you delimit them in NonStop SQL/MX.

Examples of Identifiers

- These are regular identifiers:

  ```
  mytable
  SALES1995
  Employee_Benefits_Selections
  CUSTOMER_BILLING_INFORMATION
  ```

  Because regular identifiers are case insensitive, NonStop SQL/MX treats all these identifiers as alternative representations of `mytable`:

  ```
  mytable        MYTABLE        MyTable        mYtAbLe
  ```

- These are delimited identifiers:

  ```
  "mytable"
  "table"
  ```

```
"1995 SALES"
"CUSTOMER-BILLING-INFORMATION"
"%&*()"
```

Because delimited identifiers are case-sensitive, NonStop SQL/MX treats the identifier `"mytable"` as different from the identifiers `"MYTABLE"` or `"MyTable"`. Trailing spaces in a delimited identifier are truncated. For example, `"mytable "` is equivalent to `"mytable"`.

You can use reserved words as delimited identifiers. For example, `table` is not allowed as a regular identifier, but `"table"` is allowed as a delimited identifier.

Indexes

An index is an ordered set of pointers to rows of a table. Each index is based on the values in one or more columns. An index is stored in a key-sequenced file.

There is always a one-to-one correspondence between index rows and base table rows.

SQL/MP Indexes

Each row in an SQL/MP index contains:

- A keytag column
- The columns specified in the CREATE INDEX statement
- The primary key of the underlying table (the user-defined primary key, the SYSKEY, or a combination of the user-defined clustering key and the SYSKEY)

See Index Keys in the *SQL/MP Reference Manual.*

SQL/MX Indexes

Each row in an SQL/MX index contains:

- The columns specified in the CREATE INDEX statement
- The clustering key of the underlying table (the user-defined clustering key, the SYSKEY, or a combination of the user-defined clustering key and the SYSKEY)

An index name is an SQL identifier. Indexes have their own namespace within a schema, so an index name might be the same as a table or constraint name. However, no two indexes in a schema can have the same name.

See CREATE INDEX Statement on page 2-80 and ALTER INDEX Statement on page 2-11.

Keys

NonStop SQL/MX supports these types of keys:

Clustering Keys
First (Partition) Keys
Index Keys
Primary Keys
SYSKEYs

Clustering Keys

NonStop SQL/MX organizes records of a table or index by using a b-tree based on the "clustering key". Values of the clustering key act as logical row-ids. The set of columns that make up the clustering key must guarantee uniqueness. If necessary, to guarantee uniqueness, NonStop SQL/MX appends an additional key to the set of columns you specify to define the clustering key as shown in Table 6-1 and Table 6-2. Any table or index that enforces uniqueness must also have the property that its primary key is the same as its clustering key.

You can update any column in the table that is not part of the clustering key.

Table 6-1 compares construction of the clustering key for tables with various combinations of the STORE BY and PRIMARY KEY options.

Table 6-1. Construction of the Clustering Key (page 1 of 2)

	Primary Key Specified	**DROPPABLE Attribute**	**Clustering Key**
No STORE BY	No	Not applicable	SYSKEY
	Yes	DROPPABLE	SYSKEY Primary key enforced by unique index.
	Yes	NOT DROPPABLE	Same as primary key
STORE BY primary key	No	Not applicable	Not supported (error)
	Yes	DROPPABLE	Not supported (error)
	Yes	NOT DROPPABLE	Same as primary key

Table 6-1. Construction of the Clustering Key (page 2 of 2)

	Primary Key Specified	DROPPABLE Attribute	Clustering Key
STORE BY key column list	No	Not applicable	Key column list + SYSKEY
	Yes	DROPPABLE	Not supported (error)
	Yes	NOT DROPPABLE	If STORE BY column list is a prefix of or the same as the primary key column list, NonStop SQL/MX uses the primary key column list. Other combinations are not supported and generate errors.

Table 6-2 compares construction of the clustering key for unique and nonunique indexes.

Table 6-2. Clustering Key for Indexes

	Unique	Nonunique
Clustering key	*indexedColumns*	*indexedColumns*+ *ClusteringKeyOfTable*
Default partitioning key*	*indexedColumns*	*indexedColumns*+ *ClusteringKeyOfTable*+ SYSKEY

* The columns of the default partitioning key are also the columns that are available for partitioning using the PARTITION BY clause

First (Partition) Keys

The FIRST KEY option of the PARTITION clause specifies the beginning of the range for a range partitioned table or index partition. The FIRST KEY clause specifies the lowest values in the partition for columns stored in ascending order and the highest values in the partition for columns stored in descending order. These column values are referred to as the partitioning key.

You specify the first value allowed in the associated partition for that column of the partitioning key as a literal. If there are more storage key columns than literal items, the first key value for each remaining key column is the lowest or highest value for the data type of the column (the lowest value for an ascending column and the highest value for a descending column).

Partitioning character columns must derive from the ISO88591 character set and cannot be floating-point data columns.

Index Keys

An index is stored in a key-sequenced file. There is always a one-to-one correspondence between index rows and base table rows.

SQL/MP Index Keys

Each row in an SQL/MP index contains:

- A two-byte column called the “keytag” column
- The columns specified in the CREATE INDEX statement
- The primary key of the underlying table (the user-defined primary key, the SYSKEY, or combination of the clustering key and the SYSKEY)

For a unique index, the primary key of the index is composed of the first two of these items. The primary key of the index cannot exceed 255 bytes, but the entire row (including the primary key of the index) can contain up to 510 bytes.

For a nonunique index, the primary key of the index is composed of all three items. The primary key cannot exceed 255 bytes. Because the primary key includes all the columns in the table, each row is also limited to 255 bytes.

For varying-length character columns, the length referred to in these byte limits is the defined column length, not the stored length. (The stored length is the expanded length, which includes two extra bytes for storing the data length of the item.)

The keytag value must be unique among indexes for the table; you can specify it when you create the index with the CREATE INDEX statement, or you can allow the system to generate it for you. (System-generated keytags are sequential numbers, beginning with one. User-specified keytag values can be either two bytes of character data or a SMALLINT UNSIGNED value in the range 1 through 65535. The keytag value for the primary key is 0.)

For more information, see the *SQL/MP Reference Manual* and the *SQL/MX Query Guide*.

SQL/MX Index Keys

Each row in an SQL/MX index contains:

- The columns specified in the CREATE INDEX statement
- The clustering (primary) key of the underlying table (the user-defined clustering key, the SYSKEY, or combination of the clustering key and the SYSKEY)

For a unique index, the clustering key of the index is composed of columns specified in create index only. The clustering key of the index cannot exceed 2010 bytes for 4K blocks and 2048 bytes for 32K blocks, but the entire row of the index can contain up to 4096 bytes.

For a nonunique index, the clustering key of the index is composed of columns specified in create index and the clustering key of the table. The clustering key of the index cannot exceed 2010 bytes for 4K blocks and 2048 bytes for 32K blocks. Because the entire row of the index is the clustering key of the index, the entire row of the index cannot exceed 2010 bytes for 4K blocks and 2048 bytes for 32K blocks. For more information, see Table 6-2 on page 6-61.

For varying-length character columns, the length referred to in these byte limits is the defined column length, not the stored length. (The stored length is the expanded length, which includes two extra bytes for storing the data length of the item.)

See CREATE INDEX Statement on page 2-80 and ALTER INDEX Statement on page 2-11.

Primary Keys

A primary key is the column or set of columns that define the uniqueness constraint for a table. The columns cannot contain nulls, and there is only one primary key constraint on a table.

SYSKEYs

A SYSKEY (or system-defined clustering key) is a clustering or storage key defined by NonStop SQL/MX rather than by the user. Tables stored in files or in key-sequenced files without a user-defined clustering key have a clustering key defined by NonStop SQL/MX and stored in a column named SYSKEY. Its type is LARGEINT SIGNED.

To establish the clustering key, in some cases NonStop SQL/MX appends a SYSKEY to ensure uniqueness. See Table 6-1 on page 6-60 and Table 6-2 on page 6-61 for the cases in which a SYSKEY is appended.

When you insert a record in a table stored in a file or in a key-sequenced file with a SYSKEY column, the file system automatically generates a value for the SYSKEY column. You cannot supply the value.

Selecting SYSKEY

You cannot update values in the SYSKEY column of any table, but you can use the SELECT statement to query SYSKEY values. If SYSKEY is provided in the value list or for a query, the value range allowed is 0 through 2**63 -1 (approximately 9.223 times 10 to the eighteenth power).

A query must explicitly select the SYSKEY column. For example, this SELECT statement does not display SYSKEY values:

```
SELECT * FROM table-name
```

However, if a view definition explicitly includes the SYSKEY column of a table, a SELECT * on the view does return SYSKEY values.

Literals

A literal is a constant you can use in an expression, in a statement, or as a parameter value. Literals are stored in columns of tables according to how you specify the column definitions in a CREATE TABLE statement. An SQL literal can be one of these data types:

Character String Literals on page 6-64	A series of characters enclosed in single quotes. Example: `'Planning'`
Datetime Literals on page 6-68	Begins with keyword DATE, TIME, or TIMESTAMP and followed by a character string. Example: `DATE '1990-01-22'`
Interval Literals on page 6-71	Begins with keyword INTERVAL and followed by a character string and an interval qualifier. Example: `INTERVAL '2-7' YEAR TO MONTH`
Numeric Literals on page 6-75	A simple numeric literal (one without an exponent) or a numeric literal in scientific notation. Example: `99E-2`

Character String Literals

Considerations for Character String Literals
SQL/MP Considerations for Character String Literals
Examples of Character String Literals

A character string literal is a series of characters enclosed in single quotes.

```
[_character-set | N]'string'
```

_character-set

specifies the character set ISO88591, UCS2, KANJI, or KSC5601. If you omit the character set specification, the default is whatever character set default you set when you installed NonStop SQL/MX. See Character Sets on page 6-4.

N

associates the system default character set with the string literal. The default is set by the value of the NATIONAL_CHARSET attribute during SQL/MX installation. See NATIONAL_CHARSET on page 10-50.

'*string*'

is a series of any input characters enclosed in single quotes. A single quote within a string is represented by two single quotes (`''`). A string can have a length of zero if you specify two single quotes (`''`) without a space in between.

You can specify string literals using hexadecimal code values in DML statements.

```
[_character-set | N] X'hex-code-value... '
| [_character-set | N] X'[space…]hex-code-value[[space…]
      hex-code-value...][space…]'
```

_character-set

specifies the character set ISO88591 or UCS2. If you omit the character set specification, the default is whatever character set default you set when you installed NonStop SQL/MX. See Character Sets on page 6-4.

N

associates the system default character set with the string literal. The default is set by the value of the NATIONAL_CHARSET attribute during SQL/MX installation. See NATIONAL_CHARSET on page 10-50.

X

represents the X in hexadecimal notation.

'*hex-code-value*'

represents the code value of a character in hexadecimal form enclosed in single quotes. It must contain an even number of hexadecimal digits. For UCS2, KANJI and KSC5601, each hex-code-value must be of four hexadecimal digits long. For ISO88591, each value must be two digits long. If *hex-code-value* is improperly formatted (for example, it contains an invalid hexadecimal digit or an odd number of hexadecimal digits), an error is returned.

space

is space sequences that can be added before or after *hex-code-value* for readability. The encoding for *space* must be the TERMINAL_CHARSET for an interactive interface and the SQL module character set for the programmatic interface.

Considerations for Character String Literals

Using String Literals

You can use a character string literal anywhere you need to supply a column value that has a character string data type. A string literal can be as long as a character column. See Character String Data Types on page 6-22.

You can also use string literals in string value expressions—for example, in expressions that use the concatenation operator (||) or in expressions that use functions returning string values.

When specifying string literals:

- Do not put a space between the character set qualifier (for example, `_KANJI`) and the character string literal (for example, `'abcd'`). If you use this character string literal in a statement, NonStop SQL/MX returns an error:

  ```
  _KANJI 'abcd'
  ```

- To specify a single quotation mark within a string literal, use two consecutive single quotation marks.
- To specify a string literal whose length is more than one line, separate the literal into several smaller string literals, and use the concatenation operator (||) to concatenate them.
- Case is significant in string literals. Lowercase letters are not equivalent to the corresponding uppercase letters.
- Leading and trailing spaces within a string literal are significant.

SQL/MP Considerations for Character String Literals

SQL/MP Stored Text With Spaces

In NonStop SQL/MX, you cannot put a space between the character set qualifier and the character string literal in a statement. For example, you must specify `_KANJI'abcd'`.

However, NonStop SQL/MP allows a space between the character set qualifier and character string literal (for example, `_KANJI 'abcd'`). When NonStop SQL/MX parses SQL/MP stored text, it accepts the space after the character set qualifier in an SQL/MP character string literal.

Inserting Into or Updating SQL/MP NCHAR Columns

NonStop SQL/MX supports inserting into or updating columns with the NCHAR data type in SQL/MP tables. The only restriction is that the NCHAR data being written to the table contains an even number of bytes.

A string literal used to insert into or update an NCHAR column in an SQL/MP table can be written:

`_UCS2'`*`string`*`'`

`_UCS2` associates the default character set with the string literal. The default is set by the value of the NATIONAL_CHARSET attribute during SQL/MX installation. See NATIONAL_CHARSET on page 10-50.

For example, suppose that column K is a UCS2 column in an SQL/MP table named T, and the NATIONAL_CHARSET is set to UCS2. This statement updates column K:

```
UPDATE T SET K = N'abcd'
```

Because the NATIONAL_CHARSET attribute is set to UCS2, the `N'abcd'` literal is a shorter way of writing `_UCS2'abcd'`:

```
UPDATE T SET K = _UCS2'abcd'
```

See NCHAR Columns in SQL/MX and SQL/MP Tables on page 6-25.

Inserting Into or Updating SQL/MP Kanji Columns

NonStop SQL/MX Release 2.x supports inserting into or updating columns with the KANJI or KSC data type in SQL/MP tables.

The only restriction is that the data being written to an SQL/MP table contains an even number of bytes. SQL/MX character functions that refer to double byte-encoded characters in KANJI and KSC5601 columns should provide the correct results. For more details, see the *SQL/MX Programming Manual for C and COBOL*.

Examples of Character String Literals

- These data type column specifications are shown with examples of literals that can be stored in the columns.

Character String Data Type	Character String Literal Example
`CHAR (12) UPSHIFT`	`'PLANNING'`
`PIC X (12)`	`'Planning'`
`VARCHAR (18)`	`'NEW YORK'`

- These are string literals:

```
'This is a string literal.'
'abc^&*'
'1234.56'
'This literal contains '' a single quotation mark.'
```

- This is a string literal concatenated over three lines:

```
'This MXCI literal is' ||
' in three parts,' ||
'specified over three lines.'
```

- This is a hexadecimal string literal representing the VARCHAR pattern of the ASCII string 'Strauß':

```
_ISO88591 X'53 74 72 61 75 DF'
```

- This is a KANJI example for the full-width character string 'ABC':

```
_kanji x'8261 8262 8263'
```

- This is a KSC5601 example for the full-width character string 'ABC':

```
-ksc5601 x'A3C1 A3C2 A3C3'
```

Datetime Literals

[SQL/MP Considerations for Datetime Literals](#)
[Examples of Datetime Literals](#)

A datetime literal is a DATE, TIME, or TIMESTAMP constant you can use in an expression, in a statement, or as a parameter value. Datetime literals have the same range of valid values as the corresponding datetime data types. You cannot use leading or trailing spaces within a datetime string (within the single quotes).

A datetime literal begins with the `DATE`, `TIME`, or `TIMESTAMP` keyword and can appear in default, USA, or European format.

```
DATE 'date' | TIME 'time' | TIMESTAMP 'timestamp'

date is:
     yyyy-mm-dd                                    Default
   | mm/dd/yyyy                                    USA
   | dd.mm.yyyy                                    European

time is:
     hh:mm:ss.msssss                               Default
   | hh:mm:ss.msssss [am | pm]                     USA
   | hh.mm.ss.msssss                               European

timestamp is:
     yyyy-mm-dd hh:mm:ss.msssss                    Default
   | mm/dd/yyyy hh:mm:ss.msssss [am | pm]  USA
   | dd.mm.yyyy hh.mm.ss.msssss                   European
```

`date,time,timestamp`

specify the datetime literal strings whose component fields are:

`yyyy`	Year, from 0001 to 9999
`mm`	Month, from 01 to 12
`dd`	Day, from 01 to 31
`hh`	Hour, from 00 to 23
`mm`	Minute, from 00 to 59
`ss`	Second, from 00 to 59
`msssss`	Microsecond, from 000000 to 999999
`am`	AM or am, indicating time from midnight to before noon
`pm`	PM or pm, indicating time from noon to before midnight

SQL/MP Considerations for Datetime Literals

Inserting Into or Updating Any SQL/MP DATETIME Column

NonStop SQL/MX supports inserting into or updating any columns with the DATETIME data type in SQL/MP tables except those consisting of FRACTION only.

Use a special SQL/MX DATETIME literal to insert into or update a DATETIME column in an SQL/MP table. The literal is written:

```
DATETIME 'datetime' [start-field TO] end-field
```

The string literal '*datetime*' is a subset of the standard datetime form:

```
'yyyy-mm-dd:hh:mm:ss.msssss'
```

The literal is followed by the qualifier, consisting of an optional start field and an end field. The qualifier has a range of logically contiguous fields in this order: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION. NonStop SQL/MX supports all SQL/MP DATETIME literals except those consisting of FRACTION only.

Nonstandard Datetime Literal Fields

NonStop SQL/MX requires that the individual fields of a DATETIME literal have the specified standard lengths. For example, this literal is not supported because the hour field is not two digits:

```
TIME '1:40:05'
```

For NonStop SQL/MX, use:

```
TIME '01:40:05'
```

Inserting Into or Updating Supported DATETIME Columns

Suppose that an SQL/MP table has a DATETIME column defined as:

```
MPDateTimeCol  DATETIME MONTH TO DAY
                   DEFAULT DATETIME '03-12' MONTH TO DAY
```

You can insert into this column by using a DATETIME MONTH TO DAY literal. For example:

```
INSERT INTO MPTable (MPDateTimeCol)
   VALUES (DATETIME '04-15' MONTH TO DAY);
```

FRACTION-Only DATETIME Columns

Suppose that an SQL/MP table has a DATETIME column defined as:

```
MPDateTimeCol  DATETIME FRACTION(6)
                   DEFAULT DATETIME '123456' FRACTION(6)
```

You cannot insert into tables with unsupported FRACTION-only DATETIME columns because you cannot specify values for these columns. Therefore, tables with columns of this type must be populated by using NonStop SQL/MP instead of NonStop SQL/MX.

You can select data from a DATETIME column. See Selecting DATETIME Columns in SQL/MP Tables on page 6-28.

Examples of Datetime Literals

- These are DATE literals in default, USA, and European formats, respectively:

```
DATE '1990-01-22'
DATE '01/22/1990'
DATE '22.01.1990'
```

- These are TIME literals in default, USA, and European formats, respectively:

```
TIME '13:40:05'
TIME '01:40:05 PM'
TIME '13.40.05'
```

- These are TIMESTAMP literals in default, USA, and European formats, respectively:

```
TIMESTAMP '1990-01-22 13:40:05'
TIMESTAMP '01/22/1990 01:40:05 PM'
TIMESTAMP '22.01.1990 13.40.05'
```

Interval Literals

[Considerations for Interval Literals](#)
[SQL/MP Considerations for Interval Literals](#)
[Examples of Interval Literals](#)

An interval literal is a constant of data type INTERVAL that represents a positive or negative duration of time as a year-month or day-time interval; it begins with the keyword INTERVAL optionally preceded or followed by a minus sign (for negative duration). You cannot include leading or trailing spaces within an interval string (within single quotes).

```
[-]INTERVAL [-]{'year-month' | 'day:time'} interval-qualifier

year-month is:
   years [-months] | months

day:time is:
     days [[:]hours [:minutes [:seconds [.fraction]]]]
   | hours [:minutes [:seconds [.fraction]]]
   | minutes [:seconds [.fraction]]
   | seconds [.fraction]

interval-qualifier is:
   start-field TO end-field | single-field

start-field is:
   {YEAR | MONTH | DAY | HOUR | MINUTE} [(leading-precision)]

end-field is:
   YEAR | MONTH | DAY | HOUR | MINUTE | SECOND [(fractional-
                                                 precision)]

single-field is:
   start-field | SECOND [(leading-precision,
                          fractional-precision)]
```

start-field TO *end-field*

must be either year-month or day-time. The *start-field* you specify must precede the *end-field* you specify in the list of field names.

{YEAR | MONTH | DAY | HOUR | MINUTE} [(*leading-precision*)]

specifies the *start-field*. A *start-field* can have a *leading-precision* up to 18 digits (the maximum depends on the number of fields in the interval). The *leading-precision* is the number of digits allowed in the *start-field*. The default for *leading-precision* is 2.

`YEAR | MONTH | DAY | HOUR | MINUTE | SECOND [(`*`fractional-precision`*`)]`

specifies the *end-field*. If the *end-field* is SECOND, it can have a *fractional-precision* up to 6 digits. The *fractional-precision* is the number of digits of precision after the decimal point. The default for *fractional-precision* is 6.

start-field `| SECOND [(`*leading-precision, fractional-precision*`)]`

specifies the *single-field*. If the *single-field* is SECOND, the *leading-precision* is the number of digits of precision before the decimal point, and the *fractional-precision* is the number of digits of precision after the decimal point.

The default for *leading-precision* is 2, and the default for *fractional-precision* is 6. The maximum for *leading-precision* is 18, and the maximum for *fractional-precision* is 6.

See Interval Data Types on page 6-31 and Interval Value Expressions on page 6-47.

'*year-month*' | '*day:time*'

specifies the date and time components of an interval literal. The day and hour fields can be separated by a space or a colon. The interval literal strings are:

years	Unsigned integer that specifies a number of years. *years* can be up to 18 digits, or 16 digits if *months* is the end-field. The maximum for the *leading-precision* is specified within the interval qualifier by either YEAR(18) or YEAR(16) TO MONTH.
months	Unsigned integer that specifies a number of months. Used as a starting field, *months* can have up to 18 digits. The maximum for the *leading-precision* is specified by MONTH(18). Used as an ending field, the value of *months* must be in the range 0 to 11.
days	Unsigned integer that specifies number of days. *days* can have up to 18 digits if there is no end-field; 16 digits if *hours* is the end-field; 14 digits if *minutes* is the end-field; and 13-*f* digits if *seconds* is the end-field, where *f* is the *fraction* less than or equal to 6. These maximums are specified by DAY(18), DAY(16) TO HOUR, DAY(14) TO MINUTE, and DAY(13-*f*) TO SECOND(*f*).
hours	Unsigned integer that specifies a number of hours. Used as a starting field, *hours* can have up to 18 digits if there is no end-field; 16 digits if *minutes* is the end-field; and 14-*f* digits if *seconds* is the end-field, where *f* is the *fraction* less than or equal to 6. These maximums are specified by HOUR(18), HOUR(16) TO MINUTE, and HOUR(14-*f*) TO SECOND(*f*). Used as an ending field, the value of *hours* must be in the range 0 to 23.

minutes Unsigned integer that specifies a number of minutes. Used as a starting field, *minutes* can have up to 18 digits if there is no end-field; and 16-*f* digits if *seconds* is the end-field, where *f* is the *fraction* less than or equal to 6. These maximums are specified by MINUTE(18), and MINUTE(16-*f*) TO SECOND(*f*). Used as an ending field, the value of *minutes* must be in the range 0 to 59.

seconds Unsigned integer that specifies a number of seconds. Used as a starting field, *seconds* can have up to 18 digits, minus the number of digits *f* in the *fraction* less than or equal to 6. This maximum is specified by SECOND(18-*f*, *f*). The value of *seconds* must be in the range 0 to 59.9(*n*), where *n* is the number of digits specified for seconds precision.

fraction Unsigned integer that specifies a fraction of a second. When *seconds* is used as an ending field, *fraction* is limited to the number of digits specified by the *fractional-precision* field following the SECOND keyword.

Considerations for Interval Literals

Length of Year-Month and Day-Time Strings

An interval literal can contain a maximum of 18 digits, in the string following the INTERVAL keyword, plus a hyphen (-) that separates the year-month fields, and colons (:) that separate the day-time fields. You can also separate day and hour with a space.

SQL/MP Considerations for Interval Literals

SQL/MP Interval Literals With Negative Durations

NonStop SQL/MX allows you to specify a negative interval by placing the sign before the entire literal, such as `-INTERVAL '5' DAY`, or immediately before the duration enclosed in quotes, such as `INTERVAL -'5' DAY`.

NonStop SQL/MX does not allow your application or SQL/MP stored text (in views, constraints, column defaults, or partitioning keys) to contain other notations of negative intervals, such as `INTERVAL '-5' DAY`.

Inserting Into or Updating Any SQL/MP INTERVAL Column

NonStop SQL/MX supports inserting into or updating any columns with the INTERVAL data type in SQL/MP tables—except those consisting of FRACTION only. Use the usual SQL/MX INTERVAL literal to insert into or update an INTERVAL column in an SQL/MP table.

Updating Supported INTERVAL Columns

Suppose that an SQL/MP table has an INTERVAL column defined as:

```
MPIntervalCol  INTERVAL YEAR TO MONTH
                   DEFAULT INTERVAL '01-03' YEAR TO MONTH
```

You can insert into this column by using an INTERVAL YEAR TO MONTH literal. For example:

```
INSERT INTO MPTable (MPIntervalCol)
   VALUES (INTERVAL '01-03' YEAR TO MONTH);
```

Updating INTERVAL SECOND TO FRACTION Columns

Suppose that an SQL/MP table has an INTERVAL column defined as:

```
MPIntervalCol  INTERVAL SECOND TO FRACTION(1)
  DEFAULT INTERVAL '30.0' SECOND TO FRACTION(1)
```

You can insert into this column by using the equivalent SQL/MX INTERVAL SECOND literal. For example:

```
INSERT INTO MPTable (MPIntervalCol)
   VALUES (INTERVAL '36.3' SECOND(2,1));
```

See SQL/MP INTERVAL SECOND TO FRACTION Types on page 6-74.

FRACTION-Only INTERVAL Columns

Suppose that an SQL/MP table has an INTERVAL column defined as:

```
MPIntervalCol  INTERVAL FRACTION(6)
                   DEFAULT INTERVAL '123456' FRACTION(6)
```

You cannot insert into tables with unsupported FRACTION-only INTERVAL columns because you cannot specify values for these columns. Therefore, you must populate tables with columns of this type by using SQL/MP instead of NonStop SQL/MX.

You can select data from an INTERVAL column. See Selecting INTERVAL Columns in SQL/MP Tables on page 6-33.

SQL/MP INTERVAL SECOND TO FRACTION Types

You must use the equivalent SQL/MX INTERVAL SECOND literal to insert into or update an SQL/MP INTERVAL SECOND TO FRACTION column. The equivalent mappings are:

SQL/MP Start Field	SQL/MP End Field	Equivalent SQL/MX Type
SECOND	SECOND or none	SECOND(2,0)
SECOND(x)	SECOND or none	SECOND(x,0)
SECOND	FRACTION	SECOND(2,6)
SECOND(x)	FRACTION	SECOND(x,6)
SECOND(x)	FRACTION(y)	SECOND(x,y)
SECOND	FRACTION(y)	SECOND(2,y)

SQL/MP Start Field	SQL/MP End Field	Equivalent SQL/MX Type
FRACTION	FRACTION	None
FRACTION(x)	FRACTION	None
FRACTION(x)	FRACTION(y)	None
FRACTION	FRACTION(y)	None

Note that in both NonStop SQL/MX and NonStop SQL/MP, the default leading precision for seconds is 2, and the default trailing precision for fraction of a second is 6.

Examples of Interval Literals

`INTERVAL '1' MONTH`	Interval of 1 month
`INTERVAL '7' DAY`	Interval of 7 days
`INTERVAL '2-7' YEAR TO MONTH`	Interval of 2 years, 7 months
`INTERVAL '5:2:15:36.33' DAY TO SECOND(2)`	Interval of 5 days, 2 hours, 15 minutes, and 36.33 seconds
`INTERVAL - '5' DAY`	Interval that subtracts 5 days
`INTERVAL '100' DAY(3)`	Interval of 100 days. This example requires an explicit leading precision of 3 because the default is 2.
`INTERVAL '364 23' DAY(3) TO HOUR`	Interval of 364 days, 23 hours. The separator for the day and hour fields can be a space or a colon.

Numeric Literals

A numeric literal represents a numeric value. Numeric literals can be represented as an exact numeric literal (without an exponent) or as an approximate numeric literal by using scientific notation (with an exponent).

```
exact-numeric-literal is:
     [+|-]unsigned-integer[.[unsigned-integer]]
   | [+|-].unsigned-integer

approximate-numeric-literal is:
   mantissa{E|e}exponent

mantissa is:
   exact-numeric-literal

exponent is:
   [+|-]unsigned-integer

unsigned-integer is:
   digit...
```

exact-numeric-literal

is an exact numeric value that includes an optional plus sign (+) or minus sign (-), up to 128 digits (0 through 9), and an optional period (.) that indicates a decimal point. Leading zeros do not count toward the 128-digit limit; trailing zeros do.

A numeric literal without a sign is a positive number. An exact numeric literal that does not include a decimal point is an integer. Every exact numeric literal has the data type NUMERIC and the minimum precision required to represent its value.

approximate-numeric-literal

is an exact numeric literal followed by an exponent expressed as an uppercase E or lowercase e followed by an optionally signed integer.

Numeric values expressed in scientific notation are treated as data type REAL if they include no more than seven digits before the exponent, but treated as type DOUBLE PRECISION if they include eight or more digits. Because of this factor, trailing zeros after a decimal can sometimes increase the precision of a numeric literal used as a DOUBLE PRECISION value.

For example, if XYZ is a table that consists of one DOUBLE PRECISION column, the inserted value:

```
INSERT INTO XYZ VALUES (1.00000000E-10)
```

has more precision than:

```
INSERT INTO XYZ VALUES (1.0E-10)
```

Examples of Numeric Literals

- These are all numeric literals, along with their display format:

Literal	**Display Format in MXCI**
477	477
580.45	580.45
+005	5
-.3175	-.3175
1300000000	1300000000
99.	99
-0.123456789012345678	-.123456789012345678
99E-2	9.9000000E-001
12.3e+5	1.2299999E+006

MXCI Parameters

Examples of MXCI Parameters

Typically, you use parameters (both within MXCI and in embedded SQL) so that you can prepare an SQL statement and then execute it later, providing different values for each execution. Within an MXCI file to be obeyed, you can also use parameters for values so that an SQL statement within the file can execute with different values.

MXCI Named Parameters

You specify a named parameter in a DML statement or a CALL statement within MXCI as:

?*param-name*

An MXCI *param-name* is preceded by a question mark. It begins with an alphabetic or underscore character and can contain up to 128 alphabetic, numeric, and underscore characters. Parameter names are case-sensitive. For example, the parameter `?pn` is not equivalent to the parameter `?PN`.

Unlike SQL identifiers, you cannot delimit MXCI parameter names with double-quote characters ("). You can use reserved words. For example, you can use `?at` as an MXCI parameter.

The value of a named parameter is set by using the SET PARAM command.

MXCI Unnamed Parameters

You specify an unnamed parameter in a DML statement or a CALL statement within MXCI as:

?

The value of an unnamed parameter is set by using the USING clause of the EXECUTE statement.

Type Assignment for Parameters

The data type of a parameter is either numeric or character. Only character literals associated with ISO88591 can be used for MXCI parameters. If the data type of the target column is either datetime or interval, you must convert (by using CAST) the parameter to the data type of the target column.

- If the parameter and target column has numeric data type, NonStop SQL/MX treats the parameter as DECIMAL(*n*), where *n* is the number of digits in the parameter value.
- If the parameter and target column has character data type, NonStop SQL/MX treats the parameter as CHAR(*n*), where *n* is the number of bytes in the parameter value.

- If the parameter is character and the target column has datetime data type, you must CAST the parameter to have the same data type as the target column.
- If the parameter is character or numeric and the target column has INTERVAL data type, you must CAST the parameter to have the INTERVAL data type.

Working With MXCI Parameters

Use these statements with MXCI parameters:

SET PARAM Command on page 4-63	Sets the value of an MXCI-named parameter.
RESET PARAM Command on page 4-60	Clears all named parameter values or a specified named parameter value.
SHOW PARAM Command on page 4-73	Displays all named parameters and their values that are defined in the current MXCI session.
EXECUTE Statement on page 2-201	Executes an SQL statement previously compiled by the PREPARE statement. You can specify values for unnamed parameters in the SQL statement with the USING clause of the EXECUTE statement.
An operation is a postfix merge if the range of data ends at the bottom of the partition. You can specify only the TO NEXT PARTITION clause. The split partition cannot be the last partition (the rightmost partition in the list). on page 2-279	Compiles an SQL statement for later execution with EXECUTE. The SQL statement might include named or unnamed parameters.

Use of Parameter Names

Each occurrence of the same parameter name within an MXCI session refers to the same parameter. The parameter has the value set by the most recent execution of a SET PARAM statement. If no SET PARAM statement has been issued, the parameter is undefined and NonStop SQL/MX returns an error message if you attempt to execute a DML statement that uses the parameter name.

Caution. If you use the same parameter name more than once in a single statement, do so carefully to avoid the loss of data in certain cases. NonStop SQL/MX considers each reference to point to the same parameter and assigns each occurrence the same data type and length as the first occurrence.

For example, during the execution of an INSERT statement, a parameter is assigned the same attributes as the column into which the parameter's value is first inserted. If NonStop SQL/MX truncates the parameter value to fit into the column, other occurrences of the parameter also receive the truncated value, even if the columns for those parameters are large enough to hold the entire value.

Examples of MXCI Parameters

- The PROJECT table has a START_DATE column. This UPDATE statement uses the character literal in the ?STARTDAY parameter to set the START_DATE column value in the PROJECT table:

```
SET PARAM ?STARTDAY '1999-11-15';
UPDATE persnl.project
  SET start_date = CAST(?STARTDAY AS DATE);
```

- Suppose that the PROJECT table has an EST_COMPLETE column whose default value is INTERVAL '30' DAY. This UPDATE statement uses the numeric literal in the ?EST parameter to update the EST_COMPLETE column value in the PROJECT table:

```
SET PARAM ?EST 60;
UPDATE persnl.project
  SET est_complete = CAST(?EST AS INTERVAL DAY);
```

Null

Null is a special symbol, independent of data type, that represents an unknown. The SQL/MX keyword NULL represents null. Null indicates that an item has no value. For sorting purposes, null is greater than all other values. You cannot store null in a column by using either INSERT or UPDATE, unless the column allows null.

A column that allows null can be null at any row position. A nullable column has extra bytes associated with it in each row. A special value stored in these bytes indicates that the column has null for that row.

Consider these guidelines:

- Using Null Versus Default Values on page 6-80
- Defining Columns That Allow or Prohibit Null on page 6-81
- Determining Whether a Column Allows Null on page 6-81
- Null in DISTINCT, GROUP BY, and ORDER BY Clauses on page 6-82
- Null and Expression Evaluation Comparison on page 6-82

Using Null Versus Default Values

There are various scenarios in which a row in a table might contain no value for a specific column. For example:

- A database of telemarketing contacts might have null AGE fields if contacts did not provide their age.
- An order record might have a DATE_SHIPPED column empty until the order is actually shipped.
- An employee record for an international employee might not have a social security number.

You allow null in a column when you want to convey that a value in the column is either unknown (such as the age of a telemarketing contact) or not applicable (such as the social security number of an international employee).

In deciding whether to allow nulls or use defaults, also note:

- Nulls are not the same as blanks. Two blanks can be compared and found equal, while the result of a comparison of two nulls is indeterminate.
- Nulls are not the same as zeros. Zeros can participate in arithmetic operations, while nulls are excluded from any arithmetic operation.

Defining Columns That Allow or Prohibit Null

The CREATE TABLE and ALTER TABLE statements define the attributes for columns within tables. A column allows nulls unless the column definition includes the NOT NULL clause or the column is part of the primary key of the table.

Null is the default for a column (other than NOT NULL) unless the column definition includes either a DEFAULT clause (other than DEFAULT NULL) or the NO DEFAULT clause. The default value for a column is the value NonStop SQL/MX inserts in a row when an INSERT statement omits a value for a particular column.

Determining Whether a Column Allows Null

To determine whether a column accepts null, use the INVOKE command to list the table description and check the column definitions. See INVOKE Command on page 4-46.

This INVOKE example illustrates how to display information about whether columns allow or prohibit null. The display shows NOT NULL for columns whose definition prohibits null.

```
INVOKE PERSNL.EMPLOYEE;
-- Definition of table SAMDBCAT.PERSNL.EMPLOYEE
-- Definition current Mon Sep 22 13:44:08 1997

  (
    EMPNUM          NUMERIC(4, 0) UNSIGNED NO DEFAULT
      HEADING 'Employee/Number' NOT NULL NOT DROPPABLE
   ,FIRST_NAME      CHAR(15) DEFAULT _ISO88591' '
      HEADING 'First Name' NOT NULL NOT DROPPABLE
   ,LAST_NAME       CHAR(20) DEFAULT _ISO88591' '
      HEADING 'Last Name' NOT NULL NOT DROPPABLE
   ,DEPTNUM         NUMERIC(4, 0) UNSIGNED NO DEFAULT
      HEADING 'Dept/Num' NOT NULL NOT DROPPABLE
   ,JOBCODE         NUMERIC(4, 0) UNSIGNED DEFAULT NULL
      HEADING 'Job/Code'
   ,SALARY          NUMERIC(8, 2) UNSIGNED DEFAULT NULL
  )

--- SQL operation complete.
```

In the preceding example, the columns EMPNUM, FIRST_NAME, LAST_NAME, and DEPTNUM are defined as NOT NULL. The columns JOBCODE and SALARY are allowed to be null.

Null in DISTINCT, GROUP BY, and ORDER BY Clauses

In evaluating the DISTINCT, GROUP BY, and ORDER BY clauses, NonStop SQL/MX considers all nulls to be equal. Additional considerations for these clauses are:

DISTINCT	Nulls are considered duplicates; therefore, a result has at most one null.
GROUP BY	The result has at most one null group.
ORDER BY	Nulls are considered greater than non-null values.

Null and Expression Evaluation Comparison

Expression Type	Condition	Result
Boolean operators (AND, OR, NOT)	Either operand is null.	For AND, the result is null. For OR, the result is true if the other operand is true, or null if the other operand is null or false. For NOT, the result is null.
Arithmetic operators	Either or both operands are null.	The result is null.
NULL predicate	The operand is null.	The result is true.
Aggregate (or set) functions (except COUNT)	Some rows have null columns. The function is evaluated after eliminating nulls.	The result is null if set is empty.
COUNT(*)	The function does not eliminate nulls.	The result is the number of rows in the table whether or not the rows are null.
COUNT COUNT DISTINCT	The function is evaluated after eliminating nulls.	The result is zero if set is empty.
Comparison: =, <>, <, >, <=, >=, LIKE	Either operand is null.	The result is null.
IN predicate	Some expressions in the IN value list are null.	The result is null if all of the expressions are null.
Subquery	No rows are returned.	The result is null.

Partitions

Typically, there is a one-to-one correspondence between a table definition and a physical file. However, large tables, or tables with special performance requirements, might require partitioning into multiple physical files.

A partition is the part of a table or index that resides on a single disk volume. Each table or index consists of at least one partition. A nonpartitioned table or index consists of exactly one partition. A partitioned table or index consists of more than one partition.

You create partitions by using the PARTITION clause in an SQL/MP or SQL/MX CREATE TABLE; or CREATE INDEX statement, SQL/MP ALTER statement, or SQL/MX MODIFY utility.

Partitioning character columns must derive from the ISO88591 character set and cannot be floating-point data columns.

SQL/MP Tables

A partition name, like a table or index name, is a Guardian name. If a table or index consists of more than one partition, the subvolume and file name portions of the name of each partition must be identical. Different partitions reside on different volumes. You cannot partition key-sequenced tables stored only by the SYSKEY.

You must specify the FIRST KEY, or first possible values, for each partition of key-sequenced tables. The primary partition contains the lowest set of key values if the first column of the key is stored in ascending order or the primary partition contains the highest set of key values if the first column of the key is stored in descending order.

For a key-sequenced table, you can use the PARTONLY MOVE clause of the SQL/MP ALTER TABLE statement to break the table into partitions or to break a partition into additional partitions.

See CREATE TABLE Statement, ALTER TABLE Statement, and Partitions in the *SQL/MP Reference Manual*.

SQL/MX Tables

If an index or a table is stored by a user-specified key, either a primary key or a key column list, you can specify partitioning for the index or table.

NonStop SQL/MX supports range partitioning and hash partitioning. With range partitioning, you use a FIRST KEY definition to define key ranges for each partition, and each record is assigned to the partition whose range includes the value of its partitioning key.

With hash partitioning, SQL uses a hash function on the values of the partitioning key, and each record is assigned to a partition based on the result. Partitioning key values are distributed among all partitions in a generally balanced way. The distribution is random: some rows are assigned more partitioning key values, and some rows are assigned fewer. However, although partitioning key values are balanced among the partitions, it is possible that records are not (for example, if data is skewed within partitioning key values).

In this scenario, suppose that you have a database with 10 partitions and 1,000 unique partitioning key values. Each partition will be assigned approximately 100 partitioning key values, plus or minus. However, if one of the partitioning key values is in 20 percent of the rows and the other 999 partitioning key values are distributed evenly among the other 80 percent of the rows, one partition will be assigned at least 20 percent of the rows, twice as many than would be expected in a random distribution.

Another example is when the unique entry count of the partitioning key values is relatively small compared to the total number of partitions. In this scenario, suppose that you have a database with 10 partitions and 20 partitioning key values. You would expect that each partition would be assigned 2 partitioning key values, but because the distribution is a random distribution based on the hash value of the partitioning key, some partitions are assigned more values and others fewer. Some partitions can get 3, 4, or more partitioning key values. Other partitions can be assigned 2, 1, or no partitioning key values. Even if there are many records, some partitions could have more than twice the expected number of records, and could partitions could have no records.

You control how values are distributed with the partitioning key. In another scenario, suppose that you want to distribute a database over many partitions, based on a unique telephone number that consists of an area code, an exchange, and a number (*nnn-nnn-nnnn*). If you use the area code values as the partitioning key, the distribution will be uneven because there are not many different area code values, and the number of different values is small in relation to the number of partitions. Instead, use the entire telephone number because it has so many more unique values.

Hash partitioning enables you to maintain partitions of approximately equal size, even if you do not know range values, if you have a partitioning key that:

- Does not have much data skew.
- Has many values relative to the number of partitions. The partitioning key should have at least 50 times as many distinct values as there are partitions.

For more information, see PARTITION Clause on page 7-6 CREATE TABLE Statement on page 2-107, CREATE INDEX Statement on page 2-80, and ALTER TABLE Statement on page 2-19. For a description of this utility including details about which attributes you can set for individual partition, see MODIFY Utility on page 2-271.

Automatically Creating Partitions

MXCS and JDBC/MX users can automatically create hash-partitioned SQL/MX tables with the Partition Overlay Specification (POS) feature of the CREATE TABLE statement. NonStop SQL/MX does not support automatic creation of range-partitioned tables.

Applications can control whether POS is enabled, the number of partitions, and the physical location of the partitions.

CONTROL QUERY DEFAULT attributes determine the number and physical location of the partitions. For values and syntax of these defaults, see Partition Management on page 10-63.

For more information on this feature, see Creating Partitions Automatically on page 2-127.

Predicates

A predicate determines an answer to a question about a value or group of values. A predicate returns true, false, or, if the question cannot be answered, unknown. Use predicates within search conditions to choose rows from tables or views.

BETWEEN Predicate on page 6-85	Determines whether a sequence of values is within a range of sequences of values.
Comparison Predicates on page 6-88 (=, <>, <, >, <=, >=)	Compares the values of sequences of expressions, or compares the values of sequences of row values that are the result of row subqueries.
EXISTS Predicate on page 6-92	Determines whether any rows are selected by a subquery. If the subquery finds at least one row that satisfies its search condition, the predicate evaluates to true. Otherwise, if the result table of the subquery is empty, the predicate is false.
IN Predicate on page 6-94	Determines if a sequence of values is equal to any of the sequences of values in a list of sequences.
LIKE Predicate on page 6-97	Searches for character strings that match a pattern.
NULL Predicate on page 6-99	Determines whether all the values in a sequence of values are null.
Quantified Comparison Predicates on page 6-101 (ALL, ANY, SOME)	Compares the values of sequences of expressions to the values in each row selected by a table subquery. The comparison is quantified by ALL, SOME, or ANY.
Rowset Predicates on page 6-104	A predicate that contains a rowset expression.

See the individual entry for a predicate or predicate group.

BETWEEN Predicate

Considerations for BETWEEN
Examples of BETWEEN

The BETWEEN predicate determines whether a sequence of values is within a range of sequences of values.

```
row-value-constructor [NOT] BETWEEN
    row-value-constructor AND row-value-constructor

row-value-constructor is:
     (expression [,expression ]...)
    | row-subquery
```

row-value-constructor

specifies an operand of the BETWEEN predicate. The three operands can be either of:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless *expression* is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See Expressions on page 6-41.

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See Subquery on page 6-112.

The three *row-value-constructors* specified in a BETWEEN predicate must contain the same number of elements. That is, the number of value expressions in each list, or the number of values returned by a row subquery, must be the same.

The data types of the respective values of the three *row-value-constructors* must be comparable. Respective values are values with the same ordinal position in the two lists. See Comparable and Compatible Data Types on page 6-17.

Considerations for BETWEEN

Logical Equivalents Using AND and OR

The predicate *expr1* BETWEEN *expr2* AND *expr3* is true if and only if this condition is true:

expr2 <= *expr1* AND *expr1* <= *expr3*

The predicate *expr1* NOT BETWEEN *expr2* AND *expr3* is true if and only if this condition is true:

expr2 > *expr1* OR *expr1* > *expr3*

Descending Columns in Keys

If a clause specifies a column in a key BETWEEN *expr2* and *expr3*, *expr3* must be greater than *expr2* even if the column is specified as DESCENDING within its table definition.

Examples of BETWEEN

- This predicate is true if the total price of the units in inventory is in the range from $1,000 to $10,000:

```
qty_on_hand * price
  BETWEEN 1000.00 AND 10000.00
```

- This predicate is true if the part cost is less than $5 or more than $800:

```
partcost NOT BETWEEN 5.00 AND 800.00
```

- This BETWEEN predicate selects the part number 6400:

```
SELECT * FROM partsupp
WHERE partnum BETWEEN 6400 AND 6700
  AND partcost > 300.00 SERIALIZABLE ACCESS;

Part/Num  Supp/Num  Part/Cost     Qty/Rec
--------  --------  ------------  ----------
    6400         1        390.00          50
    6401         2        500.00          20
    6401         3        480.00          38

--- 3 row(s) selected.
```

- Find names between Jody Selby and Gene Wright:

```
(last_name, first_name) BETWEEN
  ('SELBY', 'JODY') AND ('WRIGHT', 'GENE')
```

The name Barbara Swift would meet the criteria; the name Mike Wright would not.

```
SELECT empnum, first_name, last_name
FROM persnl.employee
WHERE (last_name, first_name) BETWEEN
  ('SELBY', 'JODY') AND ('WRIGHT', 'GENE');

EMPNUM  FIRST_NAME       LAST_NAME
------  ---------------  --------------------
    43  PAUL             WINTER
    72  GLENN            THOMAS
    74  JOHN             WALKER
   ...
--- 15 row(s) selected.
```

Comparison Predicates

Considerations for Comparison Predicates
Examples of Comparison Predicates

A comparison predicate compares the values of sequences of expressions, or the values of sequences of row values that are the result of row subqueries.

```
row-value-constructor comparison-op row-value-constructor

comparison-op is:
     =    Equal
   | <>   Not equal
   | <    Less than
   | >    Greater than
   | <=   Less than or equal to
   | >=   Greater than or equal to

row-value-constructor is:
     (expression [,expression]...)
   | row-subquery
```

row-value-constructor

specifies an operand of a comparison predicate. The two operands can be either of these:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless *expression* is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See Expressions on page 6-41.

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See Subquery on page 6-112.

The two *row-value-constructors* must contain the same number of elements. That is, the number of value expressions in each list, or the number of values returned by a row subquery, must be the same.

The data types of the respective values of the two *row-value-constructors* must be comparable. (Respective values are values with the same ordinal position in the two lists.) See Comparable and Compatible Data Types on page 6-17.

Considerations for Comparison Predicates

When a Comparison Predicate Is True

NonStop SQL/MX determines whether a relationship is true or false by comparing values in corresponding positions in sequence, until it finds the first nonequal pair.

You cannot use a comparison predicate in a WHERE or HAVING clause to compare row value constructors when the value expressions in one row value constructor are equal to null. Use the IS NULL predicate instead.

Suppose that there are two rows with multiple components, X and Y:

`X=(X1,X2,...,Xn), Y=(Y1,Y2,...,Yn).`

Predicate `X=Y` is true if for all `i=1,...,n: Xi=Yi`. For this predicate, NonStop SQL/MX must look through all values. Predicate `X = Y` is false if for some `i Xi<>Yi`. When SQL finds nonequal components, it stops and does not look at remaining components.

Predicate `X<>Y` is true if `X=Y` is false. If `X1<>Y1`, NonStop SQL/MX does not look at all components. It stops and returns a value of false for the `X=Y` predicate and a value of true for the `X<>Y` predicate. Predicate `X<>Y` is false if `X=Y` is true, or for all `i=1,...,n: Xi=Yi`. In this situation, NonStop SQL/MX must look through all components.

Predicate `X>Y` is true if for some index `m Xm>Ym` and for all `i=1,..,m-1: Xi=Yi`. NonStop SQL/MX does not look through all components. It stops when it finds the first nonequal components, `Xm<>Ym`. If `Xm>Ym,` the predicate is true. Otherwise the predicate is false. The predicate is also false if all components are equal, or `X=Y`.

Predicate `X>=Y` is true if `X>Y` is true or `X=Y` is true. In this scenario, NonStop SQL/MX might look through all components and return true if they are all equal. It stops at the first nonequal components, `Xm<>Ym`. If `Xm>Ym,` the predicate is true. Otherwise, it is false.

Predicate `X<Y` is true if for some index `m Xm<Ym,` and for all `i=1,..,m-1: Xi=Yi`. NonStop SQL/MX does not look through all components. It stops when it finds the first nonequal components `Xm<>Ym`. If `Xm<Ym,` the predicate is true. Otherwise, the predicate is false. The predicate is also false if all components are equal, or `X=Y`.

Predicate `X<=Y` is true if `X<Y` is true or `X=Y` is true. In this scenario, NonStop SQL/MX might need to look through all components and return true if they are all equal. It stops at the first nonequal components, `Xm<>Ym`. If `Xm<Ym,` the predicate is true. Otherwise, it is false.

Comparing Character Data

For comparisons between character strings of different lengths, the shorter string is padded on the right with spaces (HEX 20) until it is the length of the longer string. Both fixed-length and variable-length strings are padded in this way.

For example, NonStop SQL/MX considers the string ‘JOE’ equal to a value JOE stored in a column of data type CHAR or VARCHAR of width three or more. Similarly, NonStop SQL/MX considers a value JOE stored in any column of the CHAR data type equal to the value JOE stored in any column of the VARCHAR data type.

Two strings are equal if all characters in the same ordinal position are equal. Lowercase and uppercase letters are not considered equivalent.

Comparing Numeric Data

Before evaluation, all numeric values in an expression are first converted to the maximum precision needed anywhere in the expression.

Comparing Interval Data

For comparisons of INTERVAL values, NonStop SQL/MX first converts the intervals to a common unit. If no common unit exists, NonStop SQL/MX reports an error. Two INTERVAL values must be both year-month intervals or both day-time intervals.

Comparing Multiple Values

Use multivalue predicates whenever possible; they are generally more efficient than equivalent conditions without multivalue predicates.

Examples of Comparison Predicates

- This predicate is true if the customer number is equal to 3210:

  ```
  custnum = 3210
  ```

- This predicate is true if the salary is greater than the average salary of all employees:

  ```
  salary >
    (SELECT AVG (salary) FROM persnl.employee);
  ```

- This predicate is true if the customer name is BACIGALUPI:

  ```
  custname = 'BACIGALUPI'
  ```

- This predicate evaluates to unknown for any rows in either CUSTOMER or ORDERS that contain null in the CUSTNUM column:

  ```
  customer.custnum > orders.custnum
  ```

- This predicate returns information about anyone whose name follows MOSS, DUNCAN in a list arranged alphabetically by last name and, for the same last name, alphabetically by first name:

  ```
  (last_name, first_name) > ('MOSS', 'DUNCAN')
  ```

  REEVES, ANNE meets this criteria, but MOSS, ANNE does not.

This multivalue predicate is equivalent to this condition with three comparison predicates:

```
(last_name > 'MOSS') OR
(last_name = 'MOSS' AND first_name > 'DUNCAN')
```

- Compare two datetime values START_DATE and the result of the CURRENT_DATE function:

```
START_DATE < CURRENT_DATE
```

- Compare two datetime values START_DATE and SHIP_TIMESTAMP:

```
CAST (start_date AS TIMESTAMP) < ship_timestamp
```

- Compare two INTERVAL values:

```
JOB1_TIME < JOB2_TIME
```

Suppose that JOB1_TIME, defined as INTERVAL DAY TO MINUTE, is 2 days 3 hours, and JOB2_TIME, defined as INTERVAL DAY TO HOUR, is 3 days.

To evaluate the predicate, NonStop SQL/MX converts the two INTERVAL values to MINUTE. The comparison predicate is true.

- The next examples contain a subquery in a comparison predicate. Each subquery operates on a separate logical copy of the EMPLOYEE table.

The processing sequence is outer to inner. A row selected by an outer query allows an inner query to be evaluated, and a single value is returned. The next inner query is evaluated when it receives a value from its outer query.

Find all employees whose salary is greater than the maximum salary of employees in department 1500:

```
SELECT first_name, last_name, deptnum, salary
  FROM persnl.employee
  WHERE salary > (SELECT MAX (salary)
                  FROM persnl.employee
                  WHERE deptnum = 1500);

FIRST_NAME       LAST_NAME             DEPTNUM  SALARY
---------------  --------------------  -------  -----------
ROGER            GREEN                    9000    175500.00
KATHRYN          HALL                     4000     96000.00
RACHEL           MCKAY                    4000    118000.00
THOMAS           RUDLOFF                  2000    138000.40
JANE             RAYMOND                  3000    136000.00
JERRY            HOWARD                   1000    137000.10

--- 6 row(s) selected.
```

Find all employees from other departments whose salary is less than the minimum salary of employees (not in department 1500) that have a salary greater than the average salary for department 1500:

```
SELECT first_name, last_name, deptnum, salary
FROM persnl.employee
WHERE deptnum <> 1500 AND
  salary < (SELECT MIN (salary)
            FROM persnl.employee
            WHERE deptnum <> 1500 AND
              salary > (SELECT AVG (salary)
                        FROM persnl.employee
                        WHERE deptnum = 1500));

FIRST_NAME       LAST_NAME             DEPTNUM  SALARY
---------------  --------------------  -------  -----------
JESSICA          CRINER                   3500     39500.00
ALAN             TERRY                    3000     39500.00
DINAH            CLARK                    9000     37000.00
BILL             WINN                     2000     32000.00
MIRIAM           KING                     2500     18000.00
...

--- 35 row(s) selected.
```

The first subquery of this query determines the minimum salary of employees from other departments whose salary is greater than the average salary for department 1500. The main query then finds the names of employees who are not in department 1500 and whose salary is less than the minimum salary determined by the first subquery.

EXISTS Predicate

The EXISTS predicate determines whether any rows are selected by a subquery. If the subquery finds at least one row that satisfies its search condition, the predicate evaluates to true. Otherwise, if the result table of the subquery is empty, the predicate is false.

```
[NOT] EXISTS subquery
```

subquery

specifies the operand of the predicate. A *subquery* is a query expression enclosed in parentheses. An EXISTS *subquery* is typically correlated with an outer query. See Subquery on page 6-112.

Examples of EXISTS

- Find locations of employees with job code 300:

```
SELECT deptnum, location FROM persnl.dept D
WHERE EXISTS
```

```
 (SELECT jobcode FROM persnl.employee E
  WHERE D.deptnum = E.deptnum AND jobcode = 300);

DEPTNUM  LOCATION
-------  -------------
   3000  NEW YORK
   3100  TORONTO
   3200  FRANKFURT
   3300  LONDON
   3500  HONG KONG

--- 5 row(s) selected.
```

In the preceding example, the EXISTS predicate contains a subquery that determines which locations have employees with job code 300. The subquery depends on the value of D.DEPTNUM from the outer query and must be evaluated for each row of the result table where D.DEPTNUM equals E.DEPTNUM. The column D.DEPTNUM is an example of an outer reference.

- Search for departments that have no employees with job code 420:

```
SELECT deptname FROM persnl.dept D
WHERE NOT EXISTS
  (SELECT jobcode FROM persnl.employee E
   WHERE D.deptnum = E.deptnum AND jobcode = 420);

DEPTNAME
------------
FINANCE
PERSONNEL
INVENTORY
...
--- 11 row(s) selected.
```

- Search for parts with less than 20 units in the inventory:

```
SELECT partnum, suppnum
FROM invent.partsupp PS
WHERE EXISTS
  (SELECT partnum FROM invent.partloc PL
   WHERE PS.partnum = PL.partnum AND qty_on_hand < 20);

PARTNUM  SUPPNUM
-------  -------
    212        1
    212        3
   2001        1
   2003        2
    ...

--- 18 row(s) selected.
```

IN Predicate

Considerations for IN
Examples of IN

The IN predicate determines if a sequence of values is equal to any of the sequences of values in a list of sequences. The NOT operator reverses its truth value. For example, if IN is true, NOT IN is false.

```
row-value-constructor
   [NOT] IN {table-subquery | in-value-list}

row-value-constructor is:
     (expression [,expression]...)
   | row-subquery

in-value-list is:
   (expression [,expression]...)
```

row-value-constructor

specifies the first operand of the IN predicate. The first operand can be either of:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless *expression* is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See Expressions on page 6-41.

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See Subquery on page 6-112.

table-subquery

is a subquery that returns a table (consisting of rows of columns). The table specifies rows of values to be compared with the row of values specified by the *row-value-constructor*. The number of values of the*row-value-constructor* must be equal to the number of columns in the result table of the *table-subquery*, and the data types of the values must be comparable.

in-value-list

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function defined on a column. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). In this case, the result of the *row-value-*

`constructor` is a single value. The data types of the values must be comparable. The limit for the number of expressions in the `in-value-list` is 1900.

Considerations for IN

Logical Equivalent Using ANY (or SOME)

The predicate *expr* `IN` (*expr1, expr2, ...*) is true if and only if the following predicate is true:

```
expr = ANY (expr1, expr2, ... )
```

IN Predicate Results

The IN predicate is true if and only if either of these is true:

- The result of the *row-value-constructor* (a row or sequence of values) is equal to any row of column values specified by *table-subquery*.

  Note that a table subquery is a query expression and can be specified as a form of a simple table; for example, as the VALUES keyword followed by a list of row values. See SELECT Statement on page 2-330.

- The result of the *row-value-constructor* (a single value) is equal to any of the values specified by the list of expressions *in-value-list*.

  In this case, it is helpful to think of the list of expressions as a one-column table—a special case of a table subquery. The degree of the row value constructor and the degree of the list of expressions are both one.

Comparing Character Data

Two strings are equal if all characters in the same ordinal position are equal. Lowercase and uppercase letters are not considered equivalent. For comparisons between character strings of different lengths, the shorter string is padded on the right with spaces (HEX 20) until it is the length of the longer string. Both fixed-length and varying-length strings are padded in this way.

For example, NonStop SQL/MX considers the string ‘JOE’ equal to a value JOE stored in a column of data type CHAR or VARCHAR of width three or more. Similarly, NonStop SQL/MX considers a value JOE stored in any column of the CHAR data type equal to the value JOE stored in any column of the VARCHAR data type.

Comparing Numeric Data

Before evaluation, all numeric values in an expression are first converted to the maximum precision needed anywhere in the expression.

Comparing Interval Data

For comparisons of INTERVAL values, NonStop SQL/MX first converts the intervals to a common unit. If no common unit exists, NonStop SQL/MX reports an error. Two INTERVAL values must be both year-month intervals or both day-time intervals.

Examples of IN

- Find those employees whose EMPNUM is 39, 337, or 452:

```
SELECT last_name, first_name, empnum
FROM persnl.employee
WHERE empnum IN (39, 337, 452);

LAST_NAME             FIRST_NAME       EMPNUM
--------------------  ---------------  ------
CLARK                 DINAH               337
SAFFERT               KLAUS                39
--- 2 row(s) selected.
```

- Find those items in PARTS whose part number is not in the PARTLOC table:

```
SELECT partnum, partdesc
FROM sales.parts
WHERE partnum NOT IN
  (SELECT partnum
   FROM invent.partloc);

PARTNUM  PARTDESC
-------  ------------------
    186  186 MegaByte Disk

--- 1 row(s) selected.
```

- Find those items (and their suppliers) in PARTS that have a supplier in the PARTSUPP table:

```
SELECT P.partnum, P.partdesc, S.suppnum, S.suppname
FROM sales.parts P,
  invent.supplier S
WHERE P.partnum, S.suppnum IN
  (SELECT partnum, suppnum
   FROM invent.partsupp);
```

- Find those employees in EMPLOYEE whose last name and job code match the list of last names and job codes:

```
SELECT empnum, last_name, first_name
FROM persnl.employee
WHERE (last_name, jobcode) IN
  (VALUES ('CLARK', 500), ('GREEN', 200));
```

LIKE Predicate

Considerations for LIKE
Examples of LIKE

The LIKE predicate searches for character strings that match a pattern.

```
match-value [NOT] LIKE pattern [ESCAPE esc-char-expression]
```

match-value

is a character value expression that specifies the set of strings to search for that match the *pattern*.

pattern

is a character value expression that specifies the pattern string for the search.

esc-char-expression

is a character value expression that must evaluate to a single character. The escape character value is used to turn off the special meaning of percent and underscore. See Wild Card Characters on page 6-98.

See Character Value Expressions on page 6-41.

Considerations for LIKE

Comparing the Value to the Pattern

The values you compare must be character strings. Lowercase and uppercase letters are not equivalent. To make lowercase letters match uppercase letters, use the UPSHIFT function. A blank is compared in the same way as any other character.

When a LIKE Predicate Is True

When you reference a column, the LIKE predicate is true if the *pattern* matches the column value. If the value of a column reference is null, the LIKE predicate evaluates to unknown for that row. If the values you compare are both empty strings (that is, strings of zero length), the LIKE predicate is true.

Using NOT

If you specify NOT, the predicate is true if the value you are comparing does not match any string to which you are comparing or is not the same length as any string to which you are comparing. For example, NAME NOT LIKE '_Z' is true if the string is not two characters long or the last character is not Z.

In a search condition, the predicate NAME NOT LIKE '_Z' is equivalent to NOT (NAME LIKE '_Z').

Wild Card Characters

You can look for similar values by specifying only part of the characters of *pattern* combined with these wild-card characters:

% Use a percent sign to indicate zero or more characters of any type. For example, `'%ART%'` matches `'SMART'`, `'ARTIFICIAL'`, and `'PARTICULAR'`—but not `'smart'`. The code value for `%r` for KANJI character set is 0x8193, while that for KSC5601 is 0xA3A5.

_ Use an underscore to indicate any single character. For example, `'BOO_'` matches `'BOOK'` or `'BOOR'`—but not `'BOO'`, `'BOOKLET'`, or `'book'`. The code value for _ for KANJI character set is 0x8151, while that for KSC5601 is 0xA3DF.

Escape Characters

To search for a string containing a percent sign or underscore, define an escape character (using ESCAPE *esc-char-expression*) to turn off the special meaning of percent sign and underscore.

To include a percent sign or underscore in the comparison string, type the escape character immediately preceding it. For example, to locate the value 'A_B', type:

```
NAME LIKE 'A\_B' ESCAPE '\'
```

To include the escape character itself in the comparison string, type two escape characters. For example, to locate 'A_B\C%', type:

```
NAME LIKE 'A\_B\\C\%' ESCAPE '\'
```

The escape character must precede only the percent sign, underscore, or escape character itself. For example, the pattern RA\BS is not valid if the escape character is defined to be `'\'`.

Comparing the Pattern to CHAR Columns

Columns of data type CHAR are fixed length. When a value is inserted into a CHAR column, NonStop SQL/MX pads the value in the column with blanks if necessary. The value 'JOE' inserted into a CHAR(6) column becomes 'JOE   ' (3 characters plus 3 blanks). The LIKE predicate is true only if the column value and the comparison value are the same length. The column value 'JOE   ' does not match 'JOE' but does match 'JOE%'.

Comparing the Pattern to VARCHAR Columns

Columns of variable-length character data types do not include trailing blanks unless blanks are specified when data is entered. For example, the value 'JOE' inserted in a VARCHAR(4) column is 'JOE' (with no trailing blanks). The value matches both 'JOE' and 'JOE%'.

If you cannot locate a value in a variable-length character column, it might be because trailing blanks were specified when the value was inserted into the table. For example, a value of '5MB ' (with 1 trailing blank) will not be located by LIKE '%MB' but will be located by '%MB%'.

Examples of LIKE

- Find all employee last names beginning with ZE:

  `last_name LIKE 'ZE%'`

- Find all job titles that match a specific string provided at execution time:

  `jobdesc LIKE ?SOMEJOB`

  This predicate example is a part of a prepared statement where the parameter value of SOMEJOB is provided at execution time.

- Find all part descriptions that are not 'FLOPPY_DISK':

  `partdesc NOT LIKE 'FLOPPY\_DISK' ESCAPE '\'`

  The escape character indicates that the underscore in `'FLOPPY_DISK'` is part of the string to search for, not a wild-card character.

NULL Predicate

The NULL predicate determines whether all the expressions in a sequence are null. See Null on page 6-80.

```
row-value-constructor IS [NOT] NULL

row-value-constructor is:
     (expression [,expression]...)
   | row-subquery
```

row-value-constructor

specifies the operand of the NULL predicate. The operand can be either of these:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless *expression* is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See Expressions on page 6-41.

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See Subquery on page 6-112.

If all of the expressions in the *row-value-constructor* are null, the IS NULL predicate is true. Otherwise, it is false. If none of the expressions in the *row-value-constructor* are null, the IS NOT NULL predicate is true. Otherwise, it is false.

Considerations for NULL

Summary of NULL Results

Let *rvc* be the value of the *row-value-constructor*. This table summarizes the results of NULL predicates. The degree of a *rvc* is the number of values in the *rvc*.

Expressions	***rvc* IS NULL**	***rvc* IS NOT NULL**	**NOT *rvc* IS NULL**	**NOT *rvc* IS NOT NULL**
degree 1: null	TRUE	FALSE	FALSE	TRUE
degree 1: not null	FALSE	TRUE	TRUE	FALSE
degree>1: all null	TRUE	FALSE	FALSE	TRUE
degree>1: some null	FALSE	FALSE	TRUE	TRUE
degree>1: none null	FALSE	TRUE	TRUE	FALSE

Note that the *rvc* IS NOT NULL predicate is not equivalent to NOT *rvc* IS NULL.

Examples of NULL

- Find all rows with null in the SALARY column:

```
salary IS NULL
```

- This predicate evaluates to true if the expression (PRICE + TAX) evaluates to null:

```
(price + tax) IS NULL
```

- Find all rows where both FIRST_NAME and SALARY are null:

```
(first_name, salary) IS NULL
```

Quantified Comparison Predicates

Considerations for ALL, ANY, SOME
Examples of ALL, ANY, SOME

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by a table subquery. The comparison operation is quantified by the logical quantifiers ALL, ANY, or SOME.

```
row-value-constructor comparison-op quantifier table-subquery

row-value-constructor is:
     (expression [,expression]...)
   | row-subquery

comparison-op is:
     =    Equal
   | <>   Not equal
   | !=   Not equal
   | <    Less than
   | >    Greater than
   | <=   Less than or equal to
   | >=   Greater than or equal to

quantifier is:
   ALL | ANY | SOME
```

row-value-constructor

specifies the first operand of a quantified comparison predicate. The first operand can be either of:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless *expression* is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See Expressions on page 6-41.

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See Subquery on page 6-112.

ALL

specifies that the predicate is true if the comparison is true for every row selected by *table-subquery* (or if *table-subquery* selects no rows), and specifies that the predicate is false if the comparison is false for at least one row selected.

`ANY | SOME`

specifies that the predicate is true if the comparison is true for at least one row selected by the *`table-subquery`* and specifies that the predicate is false if the comparison is false for every row selected (or if *`table-subquery`* selects no rows).

`table-subquery`

provides the values for the comparison. The number of values returned by the *`row-value-constructor`* must be equal to the number of values specified by the *`table-subquery`*, and the data types of values returned by the *`row-value-constructor`* must be comparable to the data types of values returned by the *`table-subquery`*. See Subquery on page 6-112.

Considerations for ALL, ANY, SOME

Let *R* be the result of the *`row-value-constructor`*, *T* the result of the *`table-subquery`*, and *RT* a row in *T*.

Result of *R* ***`comparison-op`*** ALL *T*

If *T* is empty or if *R `comparison-op` RT* is true for every row *RT* in *T*, the *`comparison-op`* ALL predicate is true.

If *R `comparison-op` RT* is false for at least one row *RT* in *T*, the *`comparison-op`* ALL predicate is false.

Result of *R* ***`comparison-op`*** ANY *T* or *R* ***`comparison-op`*** SOME *T*

If *T* is empty or if *R `comparison-op` RT* is false for every row *RT* in *T*, the *`comparison-op`* ANY predicate is false.

If *R `comparison-op` RT* is true for at least one row *RT* in *T*, the *`comparison-op`* ANY predicate is true.

Examples of ALL, ANY, SOME

- This predicate is true if the salary is greater than the salaries of all the employees who have a jobcode of 420:

  ```
  salary > ALL (SELECT salary
                FROM persnl.employee
                WHERE jobcode = 420)
  ```

  Consider this SELECT statement using the preceding predicate:

  ```
  SELECT empnum, first_name, last_name, salary
  FROM persnl.employee
  WHERE salary > ALL (SELECT salary
                      FROM persnl.employee
                      WHERE jobcode = 420);
  ```

The inner query providing the comparison values yields these results:

```
SELECT salary
FROM persnl.employee
WHERE jobcode = 420;

SALARY
-----------
   33000.00
   36000.00
   18000.10

--- 3 row(s) selected.
```

The SELECT statement using this inner query yields these results. The salaries listed are greater than the salary of every employees with jobcode equal to 420—that is, greater than $33,000.00, $36,000.00, and $18,000.10:

```
SELECT empnum, first_name, last_name, salary
FROM persnl.employee
WHERE salary > ALL (SELECT salary
                    FROM persnl.employee
                    WHERE jobcode = 420);

EMPNUM  FIRST_NAME       LAST_NAME             SALARY
------  ---------------  --------------------  -----------
     1  ROGER            GREEN                   175500.00
    23  JERRY            HOWARD                  137000.10
    29  JANE             RAYMOND                 136000.00
   ...
   343  ALAN             TERRY                    39500.00
   557  BEN              HENDERSON                65000.00
   568  JESSICA          CRINER                   39500.00

--- 23 row(s) selected.
```

- This predicate is true if the part number is equal to any part number with more than five units in stock:

```
partnum = ANY (SELECT partnum
               FROM sales.odetail
               WHERE qty_ordered > 5)
```

Consider this SELECT statement using the preceding predicate:

```
SELECT ordernum, partnum, qty_ordered
FROM sales.odetail
WHERE partnum = ANY (SELECT partnum
                      FROM sales.odetail
                     WHERE qty_ordered > 5);
```

The inner query providing the comparison values yields these results:

```
SELECT partnum
FROM sales.odetail
WHERE qty_ordered > 5;

Part/Num
--------
    2403
    5100
    5103
    6301
    6500
    ....

--- 60 row(s) selected.
```

The SELECT statement using this inner query yields these results. All of the order numbers listed have part number equal to any part number with more than five total units in stock—that is, equal to 2403, 5100, 5103, 6301, 6500, and so on:

```
SELECT ordernum, partnum, qty_ordered
FROM sales.odetail
WHERE partnum = ANY (SELECT partnum
                     FROM sales.odetail
                     WHERE qty_ordered > 5);

Order/Num   Part/Num  Qty/Ord
----------  --------  ----------
    100210       244           3
    100210      2001           3
    100210      2403           6
    100210      5100          10
    100250       244           4
    100250      5103          10
    100250      6301          15
    100250      6500          10
    ......      ....          ..

--- 71 row(s) selected.
```

Rowset Predicates

A predicate that contains a rowset expression is called a *rowset predicate*. A rowset predicate is an array of single value predicates, where the *n*th predicate is composed from *n*th rowset element. Each array element in a rowset predicate returns true, false or unknown.

For more information about rowsets, see the *SQL/MX Programming Manual for C and COBOL*.

Pseudocolumns

A pseudocolumn is a virtual column, which behaves like a table column. You can only select values from a pseudocolumn. You cannot insert, update, or delete values from a pseudocolumn. User queries can access the values from a sequence generator using the following pseudocolumns:

- CURRVAL, which returns the current value in the sequence
- NEXTVAL, which returns the next value in the sequence

```
sequence.CURRVAL | sequence.NEXTVAL
```

sequence

specifies the ANSI name of the sequence generator.

Considerations for Pseudocolumns

You must access NEXTVAL at least once in a session before accessing CURRVAL. You can access NEXTVAL and CURRVAL in same or different statements in a session.

You can use the pseudocolumns in:

- The select list of a SELECT statement that is not contained in the subquery, or the view
- The select list of a top level subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You must not use the pseudocolumns in:

- A trigger, view, or constraint definition
- An embedded UPDATE, DELETE, or stream access query
- A DELETE statement
- An aggregate or sequence function
- The WHERE clause of a SELECT statement
- A query with a GROUP BY, SEQUENCE BY, or ORDER BY clause
- A CASE expression
- A query with a UNION clause
- As a parameter to a CALL statement

Rules for Pseudocolumns

- After a sequence generator is created, the first reference to NEXTVAL from an SQL statement returns the initial value (START WITH value) of the sequence. Subsequent references to NEXTVAL returns the current value and then increments or decrements the sequence value by the defined increment or decrement.
- Any reference to CURRVAL always returns the current value of the sequence, which is the value returned by the last reference to NEXTVAL.
- If the SQL statement contains references to both CURRVAL and NEXTVAL, then the sequence is incremented or decremented and the same value is returned for both CURRVAL and NEXTVAL.
- The NEXTVAL value is incremented or decremented for the following:
  - Every row inserted by the outermost INSERT statement.

- Every row returned by the outermost SELECT statement of an INSERT...SELECT statement. In this case, the NEXTVAL reference must be from the VALUES clause.
- Every row projected by the outermost SELECT statement.
- Every row updated with an UPDATE statement. In this case, sequence can be acccessed only from the SET clause.

- The sequence generator is updated by an internal transaction. Sequence generator updates are independent of the current DML transaction.

Schemas

The ANSI SQL:1999 schema name is an SQL identifier that is unique for a given ANSI catalog name. NonStop SQL/MX automatically qualifies a schema name with the current default catalog name unless you explicitly specify a catalog name with the schema name:

catalog.schema

The three-part logical name of the form *catalog.schema.object* is an ANSI name. The parts *catalog* and *schema* denote the ANSI-defined catalog and schema.

To be compliant with ANSI SQL:1999, NonStop SQL/MX provides support for ANSI three-part object names. By using these names, you can develop ANSI-compliant applications that access all SQL/MX and SQL/MP objects. You can access SQL/MX objects with the three-part name of the actual object, but you must create an alias for SQL/MP objects. See CREATE SQLMP ALIAS Statement on page 2-104 and ALTER SEQUENCE Statement on page 2-13 for more information.

See SET SCHEMA Statement on page 2-370, Object Naming on page 10-60, and Using NonStop SQL/MX to Access SQL/MP Databases on page 1-25.

Search Condition

A search condition is used to choose rows from tables or views, depending on the result of applying the condition to rows. The condition is a Boolean expression consisting of predicates combined together with OR, AND, and NOT operators.

You can use a search condition in the WHERE clause of a SELECT, DELETE, or UPDATE statement, the HAVING clause of a SELECT statement, the searched form of a CASE expression, the ON clause of a SELECT statement that involves a join, a CHECK constraint, or a ROWS SINCE sequence function.

```
search-condition is:
   boolean-term | search-condition OR boolean-term

boolean-term is:
   boolean-factor | boolean-term AND boolean-factor

boolean-factor is:
   [NOT] boolean-primary

boolean-primary is:
   predicate | (search-condition)
```

`OR`

specifies the resulting search condition is true if and only if either of the surrounding predicates or search conditions is true.

`AND`

specifies the resulting search condition is true if and only if both the surrounding predicates or search conditions are true.

`NOT`

reverses the truth value of its operand—the following predicate or search condition.

predicate

is a BETWEEN, comparison, EXISTS, IN, LIKE, NULL, or quantified comparison predicate. A predicate specifies conditions that must be satisfied for a row to be chosen. See Predicates on page 6-85 and individual entries.

Considerations for Search Condition

Order of Evaluation

SQL evaluates search conditions in this order:

1. Predicates within parentheses
2. NOT
3. AND
4. OR

Column References

Within a search condition, a reference to a column refers to the value of that column in the row currently being evaluated by the search condition.

Subqueries

If a search condition includes a subquery and the subquery returns no values, the predicate evaluates to null. See Subquery on page 6-112.

Examples of Search Condition

- Select rows by using a search condition composed of three comparison predicates joined by AND operators:

```
select O.ordernum, O.deliv_date, OD.qty_ordered
FROM sales.orders O,
  sales.odetail OD
WHERE qty_ordered < 9 AND deliv_date <= DATE '1998-11-01'
  AND O.ordernum = OD.ordernum;

ORDERNUM    DELIV_DATE  QTY_ORDERED
----------  ----------  -----------
    100210  1997-04-10            3
    100210  1997-04-10            3
    100210  1997-04-10            6
    100250  1997-06-15            4
    101220  1997-12-15            3
       ...
--- 28 row(s) selected.
```

- Select rows by using a search condition composed of three comparison predicates, two of which are joined by an OR operator (within parentheses), and where the result of the OR and the first comparison predicate are joined by an AND operator:

```
SELECT partnum, S.suppnum, suppname
FROM invent.supplier S,
  invent.partsupp PS
WHERE S.suppnum = PS.suppnum
  AND (partnum < 3000 OR partnum = 7102);

PARTNUM  SUPPNUM  SUPPNAME
-------  -------  ------------------
    212        1  NEW COMPUTERS INC
    244        1  NEW COMPUTERS INC
    255        1  NEW COMPUTERS INC
    ...
   7102       10  LEVERAGE INC

--- 18 row(s) selected.
```

Rowset Search Condition

A search condition that contains a rowset predicate is a *rowset search condition*. A rowset search condition applies an array of search conditions to tables or views successively, starting from the first search condition, which is obtained from the first rowset element, and proceeding to the last search condition, which is obtained from the last rowset element. All the search conditions are applied in a single SQL statement.

You can use a rowset search condition in the:

- WHERE clause of a SELECT, DELETE, or UPDATE statement
- HAVING clause of a SELECT statement
- searched form of a CASE expression
- ON clause of a SELECT statement that involves a join

For more information about rowsets, see the *SQL/MX Programming Manual for C and COBOL*.

Sequence Generators

A sequence generator is a database object that generates unique sequential values. For example, you can use sequence generators to generate primary key values. After a number is generated, the sequence is incremented for ascending sequence and decremented for descending sequence.

For information about creating, altering and dropping sequence generators, see CREATE SEQUENCE Statement on page 2-100, ALTER SEQUENCE Statement on page 2-13, and DROP SEQUENCE Statement on page 2-185.

After a sequence generator is created, you can access its values in SQL statements with the CURRVAL pseudocolumn, which returns the current value of the sequence, or the NEXTVAL pseudocolumn, which increments or decrements the sequence and returns the new value.

For more information, see Pseudocolumns on page 6-105.

SQL/MP Aliases

In product versions prior to SQL/MX Release 2.x, you referenced SQL/MP database objects using their Guardian physical names. In SQL/MX Release 2.x you can create SQL/MP aliases that map logical object names to physical Guardian locations. SQL/MP aliases are simulated ANSI names that represent the underlying Guardian physical names of SQL/MP objects. True ANSI names do not exist for SQL/MP objects.

You can use the CREATE SQLMP ALIAS command within your application to create the mappings from logical to physical names. When NonStop SQL/MX executes this command, it inserts a mapping as a row in the OBJECTS table.

See CREATE SQLMP ALIAS Statement on page 2-104, ALTER SEQUENCE Statement on page 2-13, and DROP SQLMP ALIAS Statement on page 2-188 for descriptions of the statements that affect SQL/MP Aliases. See OBJECTS Table on page 10-22 for a description of the table. See Database Object Names on page 6-13 for a description of Object Names and their relationship with the OBJECTS Table.

Stored Procedures

A stored procedure is a type of user-defined routine (UDR) that operates within a database server. Stored procedures are registered in NonStop SQL/MX during the execution of a CREATE PROCEDURE statement and invoked by NonStop SQL/MX during the execution of a CALL statement. For more information, see CREATE PROCEDURE Statement on page 2-88 and CALL Statement on page 2-54.

Unlike a user-defined function, a stored procedure does not return a value directly to the caller. Instead, a stored procedure returns a value to a host variable or dynamic parameter in its parameter list.

NonStop SQL/MX supports only stored procedures that are written in the Java language. For more information, see the *SQL/MX Guide to Stored Procedures in Java*.

Subquery

A subquery is a query expression enclosed in parentheses. Its syntactic form is specified in the syntax of a SELECT statement. For more information about query expressions, see SELECT Statement on page 2-330.

A subquery is used to provide values for a BETWEEN, comparison, EXISTS, IN, or quantified comparison predicate in a search condition. It is also used to specify a derived table in the FROM clause of a SELECT statement.

A subquery can be a table, row, or scalar subquery. Therefore, its result table can be a table consisting of multiple rows and columns, a single row of column values, or a single row consisting of only one column value.

When you use *rowset-search-condition* in a subquery, all the individual *search-conditions* in the rowset are applied successively. The result table is the union of all the rows selected by these successive applications. Using rowsets in a subquery implies that the entire rowset in the subquery is evaluated and the result table passed on the outer query. If the outer query has *rowset-search-conditions*, for each element in the outer query r*owset-search-condition*, NonStop SQL/MX will use the entire result table from the subquery, obtained by evaluating all the search conditions in the subquery *rowset-search-condition*.

SELECT Form of a Subquery

A subquery is typically specified as a special form of a SELECT statement enclosed in parentheses that queries (or selects) to provide values in a search condition or to specify a derived table as a table reference.

The form of a subquery specified as a SELECT statement is:

```
( SELECT [ALL | DISTINCT] select-list
     FROM table-ref [,table-ref]...
     [WHERE search-condition | rowset-search-condition]
     [GROUP BY colname [,colname]...]
     [HAVING search-condition | rowset-search-condition]
     [[FOR] access-option ACCESS]
     [IN {SHARE | EXCLUSIVE} MODE]
     [UNION [ALL] select-stmt] )
```

Notice that an ORDER BY clause is not allowed in a subquery.

Using Subqueries to Provide Comparison Values

When a subquery is used to provide comparison values, the SELECT statement that contains the subquery is called an *outer query*. The subquery within the SELECT is called an *inner query*. In this case, the differences between the SELECT statement and the SELECT form of a subquery are:

- A subquery is always enclosed in parentheses.
- A subquery cannot contain an ORDER BY clause.
- If a subquery is not part of an EXISTS, IN, or quantified comparison predicate, and the subquery evaluates to more than one row, a run-time error occurs.

Nested Subqueries When Providing Comparison Values

An outer query (a main SELECT statement) can have up to 15 levels of nested subqueries. Subqueries within the same WHERE or HAVING clause are at the same level. For example, this query has one level of nesting:

```
SELECT * FROM TABLE1
  WHERE A = (SELECT P FROM TABLE2 WHERE Q = 1)
    AND B = (SELECT X FROM TABLE3 WHERE Y = 2)
```

A subquery within the WHERE clause of another subquery is at a different level, however, so this query has two levels of nesting:

```
SELECT * FROM TABLE1
  WHERE A = (SELECT P FROM TABLE2
             WHERE Q = (SELECT X FROM TABLE3
                        WHERE Y = 2))
```

Correlated Subqueries When Providing Comparison Values

In the search condition of a subquery, when you refer to columns of any table or view defined in an outer query, the reference is called an outer reference. A subquery containing an outer reference is called a correlated subquery.

If you refer to a column name that occurs in more than one outer query, you must qualify the column name with the correlation name of the table or view to which it belongs. Similarly, if you refer to a column name that occurs in the subquery and in one or more outer queries, you must qualify the column name with the correlation name of the table or view to which it belongs. The correlation name is known to other subqueries at the same level, or to inner queries but not to outer queries.

If you use the same correlation name at different levels of nesting, an inner query uses the one from the nearest outer level. MXCI checks the FROM clause of the subquery first, then its outer query, and so forth, until it determines the applicable table or view.

Tables

A table is a logical representation of data in which a set of records is represented as a sequence of rows, and the set of fields common to all rows is represented by columns. A column is a set of values of the same data type with the same definition. The intersection of a row and column represents the data value of a particular field in a particular record.

Every table must have one or more columns, but the number of rows can be zero. There is no inherent order of rows within a table.

You create an SQL/MX or SQL/MP user table by using the CREATE TABLE statement in the appropriate environment. The definition of a user table within the statement includes this information:

- Name of the table
- Name of each column of the table
- Type of data you can store in each column of the table
- Other information about the table, including the physical characteristics of the file that stores the table (for example, the storage order of rows within the table)

An SQL/MP table is described in an SQL/MP catalog and stored in a physical file in the Guardian environment. An SQL/MP table name must be a Guardian name of the form:
`[\`*`node`*`.][[$`*`volume`*`.]`*`subvol.`*`]`*`filename`*

An SQL/MX table is described in an SQL/MX schema and stored in a physical file in the Guardian environment. An SQL/MX table name can be a fully qualified ANSI name of the form `catalog-name.schema-name.object-name`.

Base Tables and Views

In some descriptions of SQL, tables created with a CREATE TABLE statement are referred to as base tables to distinguish them from views, which are referred to as logical tables.

A view is a named logical table defined by a query specification that uses one or more base tables or other views. See Views on page 6-115.

Example of a Base Table

For example, this EMPLOYEE table is a base table in the sample database:

EMPNUM	FIRST_NAME	LAST_NAME	DEPTNUM	JOBCODE	SALARY
1	ROGER	GREEN	9000	100	175500.00
23	JERRY	HOWARD	1000	100	137000.00
75	TIM	WALKER	3000	300	32000.00
. . .	. . .	. . .	. . .	. . .	. . .

In this sample table, the columns are EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, JOBCODE, and SALARY. The values in each column have the same data type.

See Tables in the *SQL/MP Reference Manual*.

Triggers

A trigger is a mechanism that resides in the database and specifies that when a particular action—an insert, delete, or update—occurs on a particular table, NonStop SQL/MX should automatically perform one or more additional actions. Triggers are not allowed on SQL/MP aliases.

For a complete description of triggers and their use, see Considerations for CREATE TRIGGER on page 2-146. See also CREATE TRIGGER Statement on page 2-144, ALTER TRIGGER Statement on page 2-48, DROP TRIGGER Statement on page 2-192, SET Statement on page 2-365, and SIGNAL SQLSTATE Statement on page 2-381.

Views

A view provides an alternate way of looking at data in one or more tables. A view is a named specification of a result table, which is a set of rows selected or generated from one or more base tables or other views. The specification is a SELECT statement that is executed whenever the view is referenced.

An view is a logical table created with the CREATE VIEW statement and derived by projecting a subset of columns, restricting a subset of rows, or both, from one or more base tables or other views.

You cannot create a view that references both an SQL/MP table and an SQL/MX table.

SQL/MX Views

The distinction between protection and shorthand views does not exist for SQL/MX views. To create a view, you must have SELECT privileges for the objects underlying the view.

A view's name must be unique among table and view names within the schema that contains it. You cannot create views with names prefixed by the name of a user metadata table. For example, you cannot create a view named HISTOGRAMS_MYVIEW.

Single table views are updatable. Multitable views are not updatable.

For more information about SQL/MX views, see CREATE VIEW Statement on page 2-154 and DROP VIEW Statement on page 2-193.

SQL/MP Views

SQL/MP views are either protection views or shorthand views. A protection view is derived from a single table and can be read, updated, and secured. A shorthand view is derived from one or more tables or other views and inherits the security of the underlying tables. A shorthand view can be read but not updated.

A view name must be a Guardian name.

For retrieval, you can use all views like base tables. Whether you can use a view in an insert, update, or delete operation depends on its definition.

For more information about SQL/MP views, see Views in the *SQL/MP Reference Manual*.

Example of a View

You can define a view to show only part of the data in a table. For example, this EMPLIST view is defined as part of the EMPLOYEE table in the sample database:

EMPNUM	FIRST_NAME	LAST_NAME	DEPTNUM	JOBCODE
1	ROGER	GREEN	9000	100
23	JERRY	HOWARD	1000	100
75	TIM	WALKER	3000	300
. . .	. . .	. . .	. . .	. . .

In this sample view, the columns are EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, and JOBCODE. The SALARY column in the EMPLOYEE table is not part of the EMPLIST view.

7 SQL/MX Clauses

Clauses are used by SQL/MX statements to specify default values, ways to sample or sort data, how to store physical data, how to partition file, and other details.

This section describes these clauses for SQL/MX objects:

DEFAULT Clause

Considerations for DEFAULT
Examples of DEFAULT

The DEFAULT option of the CREATE TABLE or ALTER TABLE *table-name* ADD COLUMN statement specifies a default value for a column being created. The default value is used when a row is inserted in the table without a value for the column.

```
DEFAULT default | NO DEFAULT | identity-column-default-
specification-type

default is:
literal
| NULL
| CURRENT_DATE
| CURRENT_TIME
| CURRENT_TIMESTAMP

identity-column-default-specification-type is:
GENERATED BY DEFAULT AS IDENTITY
| GENERATED ALWAYS AS IDENTITY
```

Syntax Description of DEFAULT

DEFAULT *literal*

is a literal of a data type compatible with the data type of the associated column.

For a character column, *literal* must be a string literal of no more than 240 characters or the length of the column, whichever is less. The maximum length of a default value for a character column is 240 bytes, which includes the control characters (character set prefixes and single quote delimiter).

For a numeric column, *literal* must be a numeric literal that does not exceed the defined length of the column. The number of digits to the right of the decimal point must not exceed the scale of the column, and the number of digits to the left of the decimal point must not exceed the number in the length (or length minus scale, if you specified scale for the column).

For a datetime column, *literal* must be a datetime literal with a precision that matches the precision of the column.

For an INTERVAL column, *literal* must be an INTERVAL literal that has the range of INTERVAL fields defined for the column.

DEFAULT NULL

specifies NULL as the default. This default can occur only with a column that allows null.

`DEFAULT CURRENT_DATE`

specifies the default value for the column as the value returned by the CURRENT_DATE function at the time of the operation that assigns a value to the column. This default can occur only with a column whose data type is DATE.

`DEFAULT CURRENT_TIME`

specifies the default value for the column as the value returned by the CURRENT_TIME function at the time of the operation that assigns a value to the column. This default can occur only with a column whose data type is TIME.

`DEFAULT CURRENT_TIMESTAMP`

specifies the default value for the column as the value returned by the CURRENT_TIMESTAMP function at the time of the operation that assigns a value to the column. This default can occur only with a column whose data type is TIMESTAMP.

`DEFAULT {CURRENT_USER | USER}`

specifies the default value for the column as the value returned by the CURRENT_USER or USER function at the time of the operation that assigns a value to the column. This default can occur only with a column whose data type is fixed or variable length CHARACTER.

`NO DEFAULT`

specifies the column has no default value. You cannot specify NO DEFAULT in an ALTER TABLE statement. See ALTER TABLE Statement on page 2-19.

`identity-column-default-specification-type`

The `identity-column-default-specification-type` option specifies if an IDENITY column is of `GENERATED BY DEFAULT AS IDENTITY` type or `GENERATED ALWAYS AS IDENTITY` type. For more details, see CREATE TABLE Statement on page 2-107.

Considerations for DEFAULT

Default Value on a CREATE TABLE Statement

When the DEFAULT clause for a column is not specified, the column definition and the NOT_NULL_CONSTRAINT_DROPPABLE_OPTION in the SYSTEM_DEFAULTS table affects the default value in these ways:

Column Definition	Default Value
column data-type	Default null.
column data-type NOT NULL DROPPABLE	Default null.

column data-type `NOT NULL NOT DROPPABLE`	No default.
column data-type `NOT NULL`	Default null when NOT_NULL_ CONSTRAINT_DROPPABLE_OPTION is set to ON.
column data-type `NOT NULL`	No default when NOT_NULL_ CONSTRAINT_DROPPABLE_OPTION is set to OFF (the default).

See CREATE TABLE Statement on page 2-107.

Examples of DEFAULT

- This example uses DEFAULT clauses on CREATE TABLE to specify default column values:

```
CREATE TABLE items
  ( item_id      CHAR(12)    NO DEFAULT
    ,description CHAR(50)    DEFAULT NULL
    ,num_on_hand INTEGER     DEFAULT 0 NOT NULL
    ,PRIMARY KEY (item_id) NOT DROPPABLE );
```

- This example uses DEFAULT clauses on CREATE TABLE to specify default column values:

```
CREATE TABLE persnl.project
  ( projcode           NUMERIC (4) UNSIGNED
                       NO DEFAULT
                       NOT NULL NOT DROPPABLE
    ,empnum            NUMERIC (4) UNSIGNED
                       NO DEFAULT
                       NOT NULL NOT DROPPABLE
    ,projdesc          VARCHAR (18)
                       DEFAULT NULL
    ,start_date        DATE
                       DEFAULT CURRENT_DATE
    ,ship_timestamp    TIMESTAMP
                       DEFAULT CURRENT_TIMESTAMP
    ,est_complete      INTERVAL DAY
                       DEFAULT INTERVAL '30' DAY
    ,PRIMARY KEY      (projcode) NOT DROPPABLE );
```

- The following example uses DEFAULT clauses on CREATE TABLE to specify default column values:

```
CREATE TABLE items
  ( item_id CHAR(12) NO DEFAULT
   ,description CHAR(50) DEFAULT NULL
   ,num_on_hand INTEGER DEFAULT 0 NOT NULL);
```

- The following example uses DEFAULT clauses on CREATE TABLE to specify default column values:

```
CREATE TABLE persnl.project
  ( projcode NUMERIC (4) UNSIGNED
    NO DEFAULT
    NOT NULL
   ,empnum NUMERIC (4) UNSIGNED
    NO DEFAULT
    NOT NULL
   ,projdesc VARCHAR (18)
    DEFAULT NULL
   ,start_date DATE
    DEFAULT CURRENT_DATE
   ,ship_timestamp TIMESTAMP
    DEFAULT CURRENT_TIMESTAMP
   ,est_complete INTERVAL DAY
    DEFAULT INTERVAL '30' DAY
   ,PRIMARY KEY (projcode)) ;
```

PARTITION Clause

Considerations for PARTITION
Examples of Partitions

The PARTITION clause of the CREATE INDEX and CREATE TABLE statements creates one or more secondary partitions for a table or index.

NonStop SQL/MX supports range partitioning and hash partitioning. See Partitions on page 6-83 for details.

PARTITION is an SQL/MX extension.

```
{[RANGE] PARTITION
  [BY (partitioning-column [,partitioning-column] ....)]
  [(ADD range-partn-defn [,ADD range-partn-defn]...)]
|
HASH PARTITION
  [BY (partitioning-column [,partitioning-column]...)]
  (ADD partn-defn [,ADD partn-defn]...)}

range-partn-defn is:
  FIRST KEY {col-value | (col-value [,col-value]...)}
  partn-defn

partn-defn is:
  LOCATION $volume[.subvolume.file-name]
  [NAME partition-name] [attribute [attribute]...]

attribute is:
    EXTENT ext-size | (pri-ext-size [,sec-ext-size])
  | MAXEXTENTS num-extents
```

`BY (partitioning-column[,partitioning-column]...)`

specifies the partitioning columns. The default is the default partitioning key described by the STORE BY clause. Partitioning character columns can only be of ISO88591 character set.

`[(ADD partn-defn [,ADD partn-defn])]`

specifies the LOCATION of secondary partitions for a range-partitioned table or a hash-partitioned table. For range-partitioned tables only, each ADD also specifies the FIRST KEY for that partition.

`FIRST KEY {col-value | (col-value [,col-value]...)}`

specifies the beginning of the range for a range-partitioned table or index partition. The FIRST KEY clause specifies the lowest values in the partition for columns stored in ascending order and the highest values in the partition for columns stored in descending order. These column values are referred to as the partitioning key.

col-value is a literal that specifies the first value allowed in the associated partition for that column of the partitioning key. If there are more storage key columns than *col-value* items, the first key value for each remaining key column is the lowest or highest value for the data type of the column (the lowest value for an ascending column and the highest value for a descending column). *col-value* must contain only characters from the ISO88591 character set.

The values you specify on the FIRST KEY clause cannot be the same as the values you specify on the FIRST KEY clause for another partition of the same table or index.

For a table partition, the values in the FIRST KEY clause have a one-to-one correspondence with the columns in the partitioning key of the table.

For an index partition, the values in the FIRST KEY clause have a one-to-one correspondence with the partitioning key of the index.

`LOCATION [\`*`node.`*`]`*`$volume`*`[.`*`subvolume`*`.`*`file-name`*`]`

specifies a disk volume and, optionally, a node, subvolume and file name for the partition. The node must be the name of a node on the Expand network. For Guardian files representing a table or index partition, a view label, or a stored procedure *node* can be any node from which the object's catalog is visible.

The subvolume must be the designated subvolume for the schema in which the table or index is being created. More than one partition of a given table or index can be located on a single disk volume.

partition-name

is an SQL identifier for a partition.

`ATTRIBUTE`

specifies attributes of the partition. See EXTENT on page 9-6 and MAXEXTENTS on page 9-7 for more information.

EXTENT	Controls the size of extents that will be allocated on disk.
MAXEXTENTS	Controls the maximum disk space to be allocated.

Considerations for PARTITION

Data Type Limitations

You cannot mix APPROXIMATE data types with EXACT data types in specifying a first key value or a default value for a column. For example, if the column has type NUMERIC (9,0), for the value of that column in a FIRST KEY clause, 1000 will be accepted, but 10E4 will not (an error is returned if 10E4 is specified in this example).

Decoupling of Clustering Key and Partitioning Key

Decoupling the clustering key from the partitioning key allows those keys to differ. NonStop SQL/MX does not support full decoupling (that is, complete independence of the keys), but does support partial decoupling in which the set of partitioning key columns is allowed to be a subset of the clustering key columns. The composition of the clustering key is described in the STORE BY clause. See the STORE BY Clause on page 7-23. The partitioning key is made up of one of these:

- The columns you specify in the PARTITION BY clause
- The clustering key (omitting SYSKEY) if no PARTITION BY clause was specified

For creation of partitioned or range partitioned tables, the set of columns you specify for the partitioning key can be identical to or a subset of the clustering key columns, excluding the SYSKEY if present, and these columns can be specified in any order. A decoupled partitioned or range partitioned index can be created.

Examples of Partitions

This example creates a table with three partitions that are on different physical volumes and which have different extent sizes:

```
CREATE TABLE TIMBUKTOO

(ORDERNUM NUMERIC    (6)   UNSIGNED NO DEFAULT NOT NULL,
 PARTNUM NUMERIC     (4)   UNSIGNED NO DEFAULT NOT NULL,
 UNIT_PRICE NUMERIC  (8,2) NO DEFAULT NOT NULL,
 QTY_ORDERED NUMERIC (5)   UNSIGNED NO DEFAULT NOT NULL,

 PRIMARY KEY (ORDERNUM, PARTNUM) NOT DROPPABLE)

 STORE BY PRIMARY KEY
 LOCATION $DATA14.ZSDLKRIS.ZZZZ0000
 ATTRIBUTE EXTENT (125000,125000) MAXEXTENTS 600

 PARTITION
 ( ADD FIRST KEY (10000) LOCATION $DATA14 EXTENT 900
   MAXEXTENTS 300,

 ADD FIRST KEY (20000) LOCATION $DATA15 EXTENT (1024,2048)
 MAXEXTENTS 600,

 ADD FIRST KEY (30000) LOCATION $DATA16 MAXEXTENTS 599
 EXTENT 66000);
```

SAMPLE Clause

Considerations for SAMPLE
Examples of SAMPLE

The SAMPLE clause of the SELECT statement specifies the sampling method used to select a subset of the intermediate result table of a SELECT statement. The intermediate result table consists of the rows returned by a WHERE clause or, if there is no WHERE clause, the FROM clause. The SAMPLE clause always uses READ UNCOMMITTED access mode. It overrides the user specified access mode. See SELECT Statement on page 2-330.

SAMPLE is an SQL/MX extension.

```
SAMPLE sampling-method

sampling-method is:
     RANDOM percent-size
   | FIRST rows-size
              [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
                [,colname [ASC[ENDING] | DESC[ENDING]]]...]
   | PERIODIC rows-size EVERY number-rows ROWS
              [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
                [,colname [ASC[ENDING] | DESC[ENDING]]]...]

percent-size is:
     percent-result PERCENT [ROWS
       | {CLUSTERS OF number-blocks BLOCKS}]
   | BALANCE WHEN condition
       THEN percent-result PERCENT [ROWS]
       [WHEN condition THEN percent-result PERCENT [ROWS]]...
       [ELSE percent-result PERCENT [ROWS]] END

rows-size is:
     number-rows ROWS
   | BALANCE WHEN condition THEN number-rows ROWS
            [WHEN condition THEN number-rows ROWS]...
            [ELSE number-rows ROWS] END
```

`RANDOM` *`percent-size`*

directs NonStop SQL/MX to choose rows randomly (each row having an unbiased probability of being chosen) without replacement from the result table. The sampling size is determined by the *`percent-size`*, defined as:

```
  percent-result PERCENT [ROWS
    | {CLUSTERS OF number-blocks BLOCKS}]
| BALANCE WHEN condition THEN percent-result PERCENT [ROWS]
    [WHEN condition THEN percent-result PERCENT [ROWS]]...
    [ELSE percent-result PERCENT [ROWS]] END
```

specifies the value of the size for RANDOM sampling by using a percent of the result table. The value `percent-result` must be a numeric literal.

You can determine the actual size of the sample. Suppose that there are *N* rows in the intermediate result table. Each row is picked with a probability of *r*%, where *r* is the sample size in PERCENT. Therefore, the actual size of the resulting sample is approximately *r*% of *N*. The number of rows picked follows a binomial distribution with mean equal to *r* * *N*/100.

If you specify a sample size greater than 100 PERCENT, NonStop SQL/MX returns all the rows in the result table plus duplicate rows. The duplicate rows are picked from the result table according to the specified sampling method. This technique is referred to as oversampling and is not allowed with cluster sampling.

`ROWS`

specifies row sampling. Row sampling is the default if you specify neither ROWS nor CLUSTERS.

`CLUSTERS OF number-blocks BLOCKS`

specifies cluster sampling. You can use the CLUSTERS clause for a base table only if there is no WHERE clause in the SELECT statement. First, a cluster is chosen randomly, and then all rows in the selected cluster are added to the sample. The size of the cluster is determined by `number-blocks`. This process is repeated until the sample size is generated. See Cluster Sampling on page 7-11.

`BALANCE`

If you specify a BALANCE expression, NonStop SQL/MX performs stratified sampling. The intermediate result table is divided into disjoint strata based on the WHEN conditions. Each stratum is sampled independently by using the sampling size. For a given row, the stratum to which it belongs is determined by the first WHEN condition that is true for that row—if there is a true condition. If there is no true condition, the row belongs to the ELSE stratum.

```
FIRST rows-size [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
                 [,colname [ASC[ENDING] | DESC[ENDING]]]...]
```

directs NonStop SQL/MX to choose the first rows from the result table. You can specify the order of the rows to sample. Otherwise, NonStop SQL/MX chooses an arbitrary order. The sampling size is determined by the `rows-size`, defined as:

```
  number-rows ROWS
| BALANCE WHEN condition THEN number-rows ROWS
        [WHEN condition THEN number-rows ROWS]...
        [ELSE number-rows ROWS] END
```

specifies the value of the size for FIRST sampling by using the number of rows intended in the sample. The value `number-rows` must be an integer literal.

You can determine the actual size of the sample. Suppose that there are *N* rows in the intermediate result table. If the size *s* of the sample is specified as a number of rows, the actual size of the resulting sample is the minimum of *s* and *N*.

```
PERIODIC rows-size EVERY number-rows ROWS
         [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
          [,colname [ASC[ENDING] | DESC[ENDING]]]...]
```

directs NonStop SQL/MX to choose the first rows from each block (or period) of contiguous rows. This sampling method is equivalent to a separate FIRST sampling for each period, and the `rows-size` is defined as in FIRST sampling.

The size of the period is specified as a number of rows. You can specify the order of the rows to sample. Otherwise, NonStop SQL/MX chooses an arbitrary order.

You can determine the actual size of the sample. Suppose that there are *N* rows in the intermediate result table. If the size *s* of the sample is specified as a number of rows and the size *p* of the period is specified as a number of rows, the actual size of the resulting sample is calculated as:

FLOOR (*N*/*p*) * *s* + minimum (MOD (*N*, *p*), *s*)

Note that minimum in this expression is used simply as the mathematical minimum of two values.

Considerations for SAMPLE

Sample Rows

In general, when you use the SAMPLE clause, the same query returns different sets of rows for each execution. The same set of rows is returned only when you use the FIRST and PERIODIC sampling methods with the SORT BY option, where there are no duplicates in the specified column combination for the sort.

Cluster Sampling

Cluster sampling is an option supported by the SAMPLE RANDOM clause in a SELECT statement. A cluster, in this sense, is a logically contiguous set of disk blocks in the file in which a table is stored. The number of blocks in a cluster is specified in the CLUSTERS subclause of the SAMPLE RANDOM clause. For example:

```
SELECT * FROM customers
SAMPLE RANDOM 1 PERCENT
CLUSTERS OF 2 BLOCKS;
```

This query randomly selects one percent of the clusters in the CUSTOMERS table and then adds each row in all selected clusters to the result table. In other words, think of the CUSTOMERS table as a sequence of disk blocks, where each two blocks in the sequence is a cluster. The preceding query selects one percent of the clusters at random and then returns all the rows in each selected cluster.

Cluster sampling can be done only on a base table, not on intermediate results.

Cluster sampling is generally faster than sampling individual rows because fewer blocks are read from disk. In random row and periodic sampling, the entire result table being sampled is read, and each row in the table is processed. In cluster sampling, only the disk blocks that are part of the result table are read and processed. Therefore, if the sampling percentage is small, the performance advantage of cluster sampling over other sampling methods can be dramatic.

Cluster sampling is designed for large tables. It might return zero rows if there are not enough blocks in a table to fill at least one cluster and you specify a large cluster size. This can also happen with a partitioned table if each partition does not have enough blocks to fill at least one cluster. For example, if a table uses 1000 blocks and is distributed over 256 partitions, there will be an average of 4 blocks per partition. If you specify a SAMPLE RANDOM clause with a cluster size of 25 blocks and ten percent, even if there are 10,000 rows in the table, sampling will result in the SELECT statement returning 0 rows. To avoid this, use a smaller CLUSTER size.

For more information, see the *SQL/MX Query Guide*.

Examples of SAMPLE

Within SQLCI, suppose that the data-mining tables SALESPER, SALES, and DEPT have been created as:

```
CREATE TABLE $db.mining.salesper
   ( empid   NUMERIC (4) UNSIGNED NOT NULL
    ,dnum    NUMERIC (4) UNSIGNED NOT NULL
    ,salary  NUMERIC (8,2) UNSIGNED
    ,age     INTEGER
    ,sex     CHAR (6)
    ,PRIMARY KEY (empid) );

CREATE TABLE $db.mining.sales
    ( empid   NUMERIC (4) UNSIGNED NOT NULL
     ,product VARCHAR (20)
     ,region  CHAR (4)
```

```
     ,amount  NUMERIC (9,2) UNSIGNED
     ,PRIMARY KEY (empid) );

CREATE TABLE $db.mining.dept
    ( dnum    NUMERIC (4) UNSIGNED NOT NULL
     ,name    VARCHAR (20)
     ,PRIMARY KEY (dnum) );
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.salesperson $db.mining.salesper;
CREATE SQLMP ALIAS db.mining.sales $db.mining.sales;
CREATE SQLMP ALIAS db.mining.department $db.mining.dept;
```

Suppose, too, that sample data is inserted into this database similar to the data in the sample database.

- Return the SALARY of the youngest 50 sales people:

```
SELECT salary
FROM salesperson
SAMPLE FIRST 50 ROWS SORT BY age;

SALARY
-----------
   90000.00
   90000.00
   28000.00
   27000.12
  136000.00
   37000.40
        ...

--- 50 row(s) selected.
```

- Return the SALARY of 50 sales people. In this case, the table is clustered on EMPID. If the optimizer chooses a plan to access rows using the primary access path, the result consists of salaries of the 50 sales people with the smallest employee identifiers.

```
SELECT salary
FROM salesperson
SAMPLE FIRST 50 ROWS;

SALARY
-----------
  175500.00
  137000.10
  136000.00
  138000.40
   75000.00
   90000.00
        ...

--- 50 row(s) selected.
```

- Return the SALARY of the youngest five sales people, skip the next 15 rows, and repeat this process until there are no more rows in the intermediate result table. Note that you cannot specify periodic sampling with the sample size larger than the period.

```
SELECT salary
FROM salesperson
SAMPLE PERIODIC 5 ROWS EVERY 20 ROWS SORT BY age;

SALARY
-----------
   90000.00
   90000.00
   28000.00
   27000.12
  136000.00
   36000.00
        ...

--- 17 row(s) selected.
```

  In this example, there are 62 rows in the SALESPERSON table. For each set of 20 rows, the first five rows are selected. The last set consists of two rows, both of which are selected.

- Compute the average salary of a random 10 percent of the sales people. You will get a different result each time you run this query because it is based on a random sample.

```
SELECT AVG(salary)
FROM salesperson
SAMPLE RANDOM 10 PERCENT;

(EXPR)
--------------------
            61928.57

--- 1 row(s) selected.
```

- Compute the average salary of a random 10 percent of the sales people using cluster sampling where each cluster is 4 blocks. You will get a different result each time you run this query because it is based on a random sample.

```
SELECT AVG(salary)
FROM salesperson
SAMPLE RANDOM 10 PERCENT CLUSTERS OF 4 BLOCKS;

(EXPR)
---------------------
            50219.524

--- 1 row(s) selected.
```

For this query execution, the number of rows returned is limited by the total number of rows in the SALESPERSON table. Therefore, it is possible that no rows are returned, and the result is null.

- This query illustrates sampling after execution of the WHERE clause has chosen the qualifying rows. The query computes the average salary of a random 10 percent of the sales people over 35 years of age. You will get a different result each time you run this query because it is based on a random sample.

```
SELECT AVG(salary)
FROM salesperson
WHERE age > 35
SAMPLE RANDOM 10 PERCENT;

(EXPR)
--------------------
            58000.00

--- 1 row(s) selected.
```

- Compute the average salary of a random 10 percent of sales people belonging to the CORPORATE department. The sample is taken from the join of the SALESPERSON and DEPARTMENT tables. You will get a different result each time you run this query because it is based on a random sample.

```
SELECT AVG(salary)
FROM salesperson S, department D
WHERE S.DNUM = D.DNUM
AND D.NAME = 'CORPORATE'
SAMPLE RANDOM 10 PERCENT;

(EXPR)
---------------------
           106250.000

--- 1 row(s) selected.
```

- In this example, the SALESPERSON table is first sampled and then joined with the DEPARTMENT table. This query computes the average salary of all the sales people belonging to the CORPORATE department in a random sample of 10 percent of the sales employees.

```
SELECT AVG(salary)
FROM ( SELECT salary, dnum
       FROM salesperson
       SAMPLE RANDOM 10 PERCENT ) AS S, department D
WHERE S.DNUM = D.DNUM
AND D.NAME = 'CORPORATE';

(EXPR)
--------------------
           37000.000

--- 1 row(s) selected.
```

Note that the results of this query and some of the results of previous queries might return null:

```
SELECT AVG(salary)
FROM ( SELECT salary, dnum
       FROM salesperson
       SAMPLE RANDOM 10 PERCENT ) AS S, department D
WHERE S.DNUM = D.DNUM
AND D.NAME = 'CORPORATE';

(EXPR)
--------------------
                   ?

--- 1 row(s) selected.
```

For this query execution, the number of rows returned by the embedded query is limited by the total number of rows in the SALESPERSON table. Therefore, it is possible that no rows satisfy the search condition in the WHERE clause.

- In this example, both the tables are sampled first and then joined. This query computes the average salary and the average sale amount generated from a random 10 percent of all the sales people and 20 percent of all the sales transactions.

```
SELECT AVG(salary), AVG(amount)
FROM ( SELECT salary, empid
       FROM salesperson
       SAMPLE RANDOM 10 PERCENT ) AS S,
     ( SELECT amount, empid
       FROM sales
       SAMPLE RANDOM 20 PERCENT ) AS T
WHERE S.empid = T.empid;

(EXPR)     (EXPR)
---------  ---------
 45000.00   31000.00

--- 1 row(s) selected.
```

- This example illustrates oversampling. This query retrieves 150 percent of the sales transactions where the amount exceeds $1000. The result contains every row at least once, and 50 percent of the rows, picked randomly, occur twice.

```
SELECT *
FROM sales
WHERE amount > 1000
SAMPLE RANDOM 150 PERCENT;

EMPID  PRODUCT               REGION  AMOUNT
-----  --------------------  ------  -----------
    1  PCGOLD, 30MB          E          30000.00
   23  PCDIAMOND, 60MB       W          40000.00
   23  PCDIAMOND, 60MB       W          40000.00
   29  GRAPHICPRINTER, M1    N          11000.00
```

```
   32  GRAPHICPRINTER, M2    S           15000.00
   32  GRAPHICPRINTER, M2    S           15000.00
  ...  ...                  ...               ...

--- 88 row(s) selected.
```

- The BALANCE option enables stratified sampling. Retrieve the age and salary of 1000 sales people such that 50 percent of the result are male and 50 percent female.

```
SELECT age, sex, salary
FROM salesperson
SAMPLE FIRST
       BALANCE WHEN sex = 'male' THEN 15 ROWS
               WHEN sex = 'female' THEN 15 ROWS
       END
ORDER BY age;

AGE          SEX     SALARY
-----------  ------  -----------
         22  male       28000.00
         22  male       90000.00
         22  female    136000.00
         22  male       37000.40
        ...  ...             ...

--- 30 row(s) selected.
```

- Retrieve all sales records with the amount exceeding $10000 and a random sample of 10 percent of the remaining records:

```
SELECT *
FROM sales
SAMPLE RANDOM
       BALANCE WHEN amount > 10000 THEN 100 PERCENT
       ELSE 10 PERCENT
       END;

EMPID  PRODUCT               REGION  AMOUNT
-----  --------------------  ------  -----------
    1  PCGOLD, 30MB          E          30000.00
   23  PCDIAMOND, 60MB       W          40000.00
   29  GRAPHICPRINTER, M1    N          11000.00
   32  GRAPHICPRINTER, M2    S          15000.00
  ...  ...                   ...             ...
  228  MONITORCOLOR, M2      N          10500.00
  ...  ...                   ...             ...

--- 32 row(s) selected.
```

- This query shows an example of stratified sampling where the conditions are not mutually exclusive:

```
SELECT *
FROM sales
SAMPLE RANDOM
BALANCE WHEN amount > 10000 THEN 100 PERCENT
        WHEN product = 'PCGOLD, 30MB' THEN 25 PERCENT
        WHEN region = 'W' THEN 40 PERCENT
        ELSE 10 PERCENT
        END;

EMPID  PRODUCT               REGION  AMOUNT
-----  --------------------  ------  -----------
    1  PCGOLD, 30MB          E          30000.00
   23  PCDIAMOND, 60MB       W          40000.00
   29  GRAPHICPRINTER, M1    N          11000.00
   32  GRAPHICPRINTER, M2    S          15000.00
   39  GRAPHICPRINTER, M3    S          20000.00
   75  LASERPRINTER, X1      W          42000.00
  ...  ...                   ...             ...

--- 30 row(s) selected.
```

SEQUENCE BY Clause

Considerations for SEQUENCE BY
Examples of SEQUENCE BY

The SEQUENCE BY clause of the SELECT statement specifies the order in which to sort the rows of the intermediate result table for calculating sequence functions. This option is used for processing time-sequenced rows in data mining applications. See SELECT Statement on page 2-330.

SEQUENCE BY is an SQL/MX extension.

```
SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
   [,colname [ASC[ENDING] | DESC[ENDING]]]...
```

colname

names a column in *select-list* or a column in a table reference in the FROM clause of the SELECT statement. *colname* is optionally qualified by a table, view, or correlation name; for example, CUSTOMER.CITY.

ASC | DESC

specifies the sort order. ASC is the default. For ordering an intermediate result table on a column that can contain null, nulls are considered equal to one another but greater than all other non-null values.

You must include a SEQUENCE BY clause if you include a sequence function in the select list of the SELECT statement. Otherwise, NonStop SQL/MX returns an error. Further, you cannot include a SEQUENCE BY clause if there is no sequence function in the select list. See Sequence Functions on page 8-7.

Considerations for SEQUENCE BY

- Sequence functions behave differently from set (or aggregate) functions and mathematical (or scalar) functions.
- If you include both SEQUENCE BY and GROUP BY clauses in the same SELECT statement, the values of the sequence functions must be evaluated first and then become input for aggregate functions in the statement.
    - For a SELECT statement that contains both SEQUENCE BY and GROUP BY clauses, you can nest the sequence function in the aggregate function:

```
SELECT ordernum,
    MAX(MOVINGSUM(qty_ordered, 3)) AS maxmovsum_qty,
    AVG(unit_price) AS avg_price
FROM odetail
SEQUENCE BY partnum
GROUP BY ordernum;
```

- To use a sequence function as a grouping column, you must use a derived table for the SEQUENCE BY query and use the derived column in the GROUP BY clause:

```
SELECT ordernum, movsum_qty, AVG(unit_price)
FROM
  (SELECT ordernum, MOVINGSUM(qty_ordered, 3), unit_price
   FROM odetail
   SEQUENCE BY partnum)
     AS tab2 (ordernum, movsum_qty, unit_price)
GROUP BY ordernum, movsum_qty;
```

- To use an aggregate function as the argument to a sequence function, you must also use a derived table:

```
SELECT MOVINGSUM(avg_price,2)
FROM
  (SELECT ordernum, AVG(unit_price)
   FROM odetail
   GROUP BY ordernum)
     AS tab2 (ordernum, avg_price)
SEQUENCE BY ordernum;
```

- Like aggregate functions, sequence functions generate an intermediate result. If the query has a WHERE clause, its search condition is applied during the generation of the intermediate result. Therefore, you cannot use sequence functions in the WHERE clause of a SELECT statement.

  - This query returns an error:

```
SELECT ordernum, partnum, RUNNINGAVG(unit_price)
FROM odetail
WHERE ordernum > 800000 AND RUNNINGAVG(unit_price) > 350
SEQUENCE BY qty_ordered;
```

  - Apply a search condition to the result of a sequence function, use a derived table for the SEQUENCE BY query, and use the derived column in the WHERE clause:

```
SELECT ordernum, partnum, runavg_price
FROM
  (SELECT ordernum, partnum, RUNNINGAVG(unit_price)
   FROM odetail
   SEQUENCE BY qty_ordered)
     AS tab2 (ordernum, partnum, runavg_price)
WHERE ordernum > 800000 AND runavg_price > 350;
```

Examples of SEQUENCE BY

- Sequentially number each row for the entire result and also number the rows for each part number:

```
SELECT RUNNINGCOUNT(*) AS RCOUNT, MOVINGCOUNT(*,
  ROWS SINCE (d.partnum<>THIS(d.partnum)))
    AS MCOUNT,
  d.partnum
FROM orders o, odetail d
WHERE o.ordernum=d.ordernum
SEQUENCE BY d.partnum, o.order_date, o.ordernum
ORDER BY d.partnum, o.order_date, o.ordernum;

RCOUNT                 MCOUNT                 Part/Num
---------------------  ---------------------  --------
                    1                      1       212
                    2                      2       212
                    3                      1       244
                    4                      2       244
                    5                      3       244
                  ...                    ...       ...
                   67                      1      7301
                   68                      2      7301
                   69                      3      7301
                   70                      4      7301

--- 70 row(s) selected.
```

- Show the orders for each date, the amount for each order item and the moving total for each order, and the running total of all the orders. The query sequences orders by date, order number, and part number. (The CAST function is used for readability only.)

```
SELECT o.ordernum,
  CAST (MOVINGCOUNT(*,ROWS SINCE(THIS(o.ordernum) <>
     o.ordernum)) AS INT) AS MCOUNT,
  d.partnum,  o.order_date,
  (d.unit_price * d.qty_ordered) AS AMOUNT,
  MOVINGSUM (d.unit_price * d.qty_ordered,
    ROWS SINCE(THIS(o.ordernum)<>o.ordernum)) AS ORDER_TOTAL,
  RUNNINGSUM (d.unit_price * d.qty_ordered) AS TOTAL_SALES
FROM orders o, odetail d
WHERE o.ordernum=d.ordernum
SEQUENCE BY o.order_date, o.ordernum, d.partnum
ORDER BY o.order_date, o.ordernum, d.partnum;

Order/Num   MCOUNT       Part/Num  Order/Date
AMOUNT      ORDER_TOTAL     TOTAL_SALES
----------  -----------  --------  ----------
----------  --------------  --------------

    100250            1       244  1997-01-23
  14000.00        14000.00        14000.00
    100250            2      5103  1997-01-23
```

```
   4000.00         18000.00         18000.00
    100250                3     6500  1997-01-23
    950.00         18950.00         18950.00

    200300                1      244  1997-02-06
  28000.00         28000.00         46950.00
    200300                2     2001  1997-02-06
  10000.00         38000.00         56950.00
    200300                3     2002  1997-02-06
  14000.00         52000.00         70950.00
       ...              ...      ...         ...
    800660               18     7102  1997-10-09
   1650.00        187360.00       1113295.00
    800660               19     7301  1997-10-09
   5100.00        192460.00       1118395.00

--- 69 row(s) selected.
```

Note that, for example, for order number 200300, the ORDER_TOTAL is a moving sum within the order date 1997-02-06, and the TOTAL_SALES is a running sum for all orders. The current window for the moving sum is defined as ROWS SINCE (THIS(o.ordernum)<>o.ordernum), which restricts the ORDER_TOTAL to the current order number.

- Show the amount of time between orders by calculating the interval between two dates:

```
SELECT RUNNINGCOUNT(*),o.order_date,DIFF1(o.order_date)
FROM orders o
SEQUENCE BY o.order_date, o.ordernum
ORDER BY o.order_date, o.ordernum ;

(EXPR)                Order/Date  (EXPR)
--------------------  ----------  -------------
                   1  1997-01-23              ?
                   2  1997-02-06             14
                   3  1997-02-17             11
                   4  1997-03-03             14
                   5  1997-03-19             16
                   6  1997-03-19              0
                   7  1997-03-27              8
                   8  1997-04-10             14
                   9  1997-04-20             10
                  10  1997-05-12             22
                  11  1997-06-01             20
                  12  1997-07-21             50
                  13  1997-10-09             80

--- 13 row(s) selected.
```

STORE BY Clause

Considerations for STORE BY

The STORE BY clause determines the order of rows within the physical file that holds the table, and has an effect on how you can partition the object.

```
STORE BY store-option

store-option is:
     PRIMARY KEY
   | (key-column-list)

key-column-list is:
   column-name [ASC[ENDING] | DESC[ENDING]]
      [,column-name [ASC[ENDING] | DESC[ENDING]]]...
```

`STORE BY store-option`

specifies a set of columns on which to base the clustering key. The clustering key determines the order of rows within the physical file that holds the table. The storage order has an effect on how you can partition the object.

store-option is defined as:

`PRIMARY KEY`

bases the clustering key on the primary key columns. This store option requires that the primary key is NOT DROPPABLE. If the primary key is defined as DROPPABLE, NonStop SQL/MX returns an error.

key-column-list

bases the clustering key on the columns in the *key-column-list*. The key columns in *key-column-list* must be specified as NOT NULL NOT DROPPABLE and cannot have a combined length of more than 247 bytes.

The default is PRIMARY KEY if you specified a PRIMARY KEY clause that has the NOT DROPPABLE constraint in the CREATE TABLE statement.

If you omit the STORE BY clause and you do not specify a PRIMARY KEY that has the NOT DROPPABLE constraint, the storage order is determined only by the SYSKEY. You cannot partition a table stored only by SYSKEY. See SYSKEYs on page 6-63.

Considerations for STORE BY

Storage Order and Partitioning

The organization of the physical files that make up a table and the order of rows within those physical files determine the ways you can partition the table and affect the performance of queries on that table.

You specify the organization and storage order with the STORE BY clause of the CREATE TABLE statement (either explicitly or by omitting the clause), and you cannot change it after the table is created. There are three possibilities.

Primary Key Storage Order

If you specify STORE BY PRIMARY KEY or you omit the STORE BY clause but specify a PRIMARY KEY clause that has the NOT DROPPABLE option, NonStop SQL/MX stores and retrieves rows in the order of the values in the primary key and allows you to partition the table based on values of the primary key.

This ordering mechanism is generally the most efficient method if you want to partition by values of a unique key.

SYSKEY Storage Order

If you omit the STORE BY clause and do not specify a PRIMARY KEY that has the NOT DROPPABLE option, NonStop SQL/MX determines the storage order for rows without reference to the data you specify for the rows.

As a mechanism for determining row order, NonStop SQL/MX creates the table with an additional column named SYSKEY (type LARGEINT SIGNED) and automatically generates a unique eight-byte number as the SYSKEY value of each row you insert in the table. Rows are stored and retrieved in ascending order by the SYSKEY value. You cannot update values in the SYSKEY column, although you can list them if you explicitly name SYSKEY in a SELECT statement. (SELECT * does not include SYSKEY.) See SYSKEYs on page 6-63.

You cannot partition a table stored only by the SYSKEY.

Key Column List Storage Order

If you specify STORE BY *key-column-list* and do not have a NOT DROPPABLE PRIMARY KEY, NonStop SQL/MX orders the table using a combination of the two methods previously described and allows you to partition based on values of the columns in *key-column-list*.

NonStop SQL/MX creates a SYSKEY column and treats it as the last column in a key that begins with the column or columns you specified in *key-column-list*. The SYSKEY column makes the overall key unique, even though the columns you specified might not be unique. NonStop SQL/MX then stores and retrieves rows in the order of the values in the overall key (the columns in *key-column-list* followed by the SYSKEY column) just as if it were a primary key.

This ordering mechanism is the only method that allows you to partition by values of a nonunique key.

You cannot specify a STORE BY key-column-list and a NOT DROPPABLE PRIMARY KEY in the same statement.

The relationship between the STORE BY clause, the primary key, and the clustering key in addition to the resulting default partitioning key is summarized in Table 6-1, Construction of the Clustering Key, on page 6-60 and Table 6-2, Clustering Key for Indexes, on page 6-61.

Effect of Storage Order on Partitioning

Primary Key Storage Order

If your clustering key is based on a non droppable primary key, you can partition the table. This ordering mechanism is generally the most efficient method if you want to partition by values of a unique key.

SYSKEY Storage Order

You cannot partition a table stored only by the SYSKEY.

Key Column List Storage Order

If you specify STORE BY key-column-list, you can partition based on values of the columns in key-column-list.

You must use this storage order to partition by values of a nonunique key.

TRANSPOSE Clause

[Considerations for TRANSPOSE](#)
[Examples of TRANSPOSE](#)

The TRANSPOSE clause of the SELECT statement generates for each row of the SELECT source table a row for each item in the transpose item list. The result table of the TRANSPOSE clause has all the columns of the source table plus, for each transpose item list, a value column or columns and an optional key column.

TRANSPOSE is an SQL/MX extension.

```
TRANSPOSE transpose-set [transpose-set]...
   [KEY BY key-colname]

transpose-set is:
   transpose-item-list AS transpose-col-list

transpose-item-list is:
     expression-list
   | (expression-list) [,(expression-list)]...

expression-list is:
   expression [,expression]...

transpose-col-list is:
     colname
   | (colname-list)

colname-list is:
   colname [,colname]...
```

transpose-item-list AS *transpose-col-list*

specifies a *transpose-set*, which correlates a *transpose-item-list* with a *transpose-col-list*. The *transpose-item-list* can be either a list of expressions or a list of expression lists enclosed in parentheses. The *transpose-col-list* can be either a single column name or a list of column names enclosed in parentheses.

For example, in the *transpose-set* TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2), the items in the *transpose-item-list* are (A,X),(B,Y), and (C,Z), and the *transpose-col-list* is (V1,V2). The number of expressions in each item must be the same as the number of value columns in the column list.

In the example TRANSPOSE A,B,C AS V, the items are A,B, and C, and the value column is V. This form can be thought of as a shorter way of writing TRANSPOSE (A),(B),(C) AS (V).

transpose-item-list

specifies a list of items. An item is either a value expression or a list of value expressions enclosed in parentheses.

expression-list

specifies a list of SQL value expressions, separated by commas. The expressions must have compatible data types.

For example, in the transpose set `TRANSPOSE A,B,C AS V`, the expressions `A,B`, and `C` have compatible data types.

(*expression-list*) [,(*expression-list*)]...

specifies a list of expressions enclosed in parentheses, followed by another list of expressions enclosed in parentheses, and so on. The number of expressions within parentheses must be equal for each list. The expressions in the same ordinal position within the parentheses must have compatible data types.

For example, in the transpose set `TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)`, the expressions `A,B`, and `C` have compatible data types, and the expressions `X,Y,` and `Z` have compatible data types.

transpose-col-list

specifies the columns that consist of the evaluation of expressions in the item list as the expressions are applied to rows of the source table.

colname

is an SQL identifier that specifies a column name. It identifies the column consisting of the values in *expression-list*.

For example, in the transpose set `TRANSPOSE A,B,C AS V,` the column `V` corresponds to the values of the expressions `A,B`, and `C`.

(*colname-list*)

specifies a list of column names enclosed in parentheses. Each column consists of the values of the expressions in the same ordinal position within the parentheses in the transpose item list.

For example, in the transpose set `TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)`, the column `V1` corresponds to the expressions `A,B`, and `C`, and the column `V2` corresponds to the expressions `X,Y,` and `Z`.

KEY BY *key-colname*

optionally specifies which expression (the value in the transpose column list corresponds to) by its position in the item list. *key-colname* is an SQL identifier. The data type of the key column is exact numeric, and the value is NOT NULL.

Considerations for TRANSPOSE

Multiple TRANSPOSE Clauses and Sets

- Multiple TRANSPOSE clauses can be used in the same query. For example:

```
SELECT KEYCOL1, VALCOL1, KEYCOL2, VALCOL2 FROM MYTABLE
TRANSPOSE A, B, C AS VALCOL1
  KEY BY KEYCOL1
TRANSPOSE D, E, F AS VALCOL2
  KEY BY KEYCOL2
```

- A TRANSPOSE clause can contain multiple transpose sets. For example:

```
SELECT KEYCOL, VALCOL1, VALCOL2 FROM MYTABLE
TRANSPOSE A, B, C AS VALCOL1
          D, E, F AS VALCOL2
  KEY BY KEYCOL
```

Degree and Column Order of the TRANSPOSE Result

The degree of the TRANSPOSE result is the degree of the source table (the result table derived from the table reference or references in the FROM clause and a WHERE clause if specified), plus one if the key column is specified, plus the cardinalities of all the transpose column lists.

The columns of the TRANSPOSE result are ordered beginning with the columns of the source table, followed by the key column if specified, and then followed by the list of column names in the order in which they are specified.

Data Type of the TRANSPOSE Result

The data type of each of the value columns is the union compatible data type of the corresponding expressions in the *transpose-item-list*. You cannot have expressions with data types that are not compatible in a *transpose-item-list*.

For example, in `TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)`, the data type of V1 is the union compatible type for A, B, and C, and the data type of V2 is the union compatible type for X, Y, and Z.

See Comparable and Compatible Data Types on page 6-17.

Cardinality of the TRANSPOSE Result

The items in each *transpose-item-list* are enumerated from 1 to N, where N is the total number of items in all the item lists in the transpose sets.

In this example with a single transpose set, the value of N is 3:

```
TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)
```

In this example with two transpose sets, the value of N is 5:

```
TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)
           L,M AS V3
```

The values 1 to N are the key values k_i. The items in each *transpose-item-list* are the expression values v_i.

The cardinality of the result of the TRANSPOSE clause is the cardinality of the source table times N, the total number of items in all the transpose item lists.

For each row of the source table and for each value in the key values k_i, the TRANSPOSE result contains a row with all the attributes of the source table, the key value k_i in the key column, the expression values v_i in the value columns of the corresponding transpose set, and NULL in the value columns of other transpose sets.

For example, consider this TRANSPOSE clause:

```
TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)
          L,M AS V3
  KEY BY K
```

The value of N is 5. One row of the SELECT source table produces this TRANSPOSE result:

columns-of-source	**K**	**V1**	**V2**	**V3**
source-row	1	*value-of-A*	*value-of-X*	NULL
source-row	2	*value-of-B*	*value-of-Y*	NULL
source-row	3	*value-of-C*	*value-of-Z*	NULL
source-row	4	NULL	NULL	*value-of-L*
source-row	5	NULL	NULL	*value-of-M*

Examples of TRANSPOSE

Suppose that MYTABLE has been created as:

```
CREATE TABLE $db.mining.mytable
 ( A INTEGER, B INTEGER, C INTEGER, D CHAR(2),
   E CHAR(2), F CHAR(2) );
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.mytable $db.mining.mytable;
```

The table MYTABLE has columns A, B, C, D, E, and F with related data. The columns A, B, and C are type INTEGER, and columns D, E, and F are type CHAR.

A	B	C	D	E	F
1	10	100	d1	e1	f1
2	20	200	d2	e2	f2

- Suppose that MYTABLE has only the first three columns: A, B, and C. The result of the TRANSPOSE clause has three times as many rows (because there are three items in the transpose item list) as there are rows in MYTABLE:

  ```
  SELECT * FROM mytable
  TRANSPOSE A, B, C AS VALCOL
    KEY BY KEYCOL;
  ```

  The result table of the TRANSPOSE query is:

A	B	C	D	E	F	KEYCOL	VALCOL
1	10	100	d1	e1	f1	1	1
1	10	100	d1	e1	f1	2	10
1	10	100	d1	e1	f1	3	100
2	20	200	d2	e2	f2	1	2
2	20	200	d2	e2	f2	2	20
2	20	200	d2	e2	f2	3	200

- This query shows that the items in the transpose item list can be any valid scalar expressions:

```
SELECT KEYCOL, VALCOL, A, B, C FROM mytable
TRANSPOSE A + B, C + 3, 6 AS VALCOL
  KEY BY KEYCOL;
```

The result table of the TRANSPOSE query is:

KEYCOL	VALCOL	A	B	C
1	11	1	10	100
2	103	1	10	100
3	6	1	10	100
1	22	2	20	200
2	203	2	20	200
3	6	2	20	200

- This query shows how the TRANSPOSE clause can be used with a GROUP BY clause. This query is typical of queries used to obtain cross-table information, where A, B, and C are the independent variables, and D is the dependent variable.

```
SELECT KEYCOL, VALCOL, D, COUNT(*) FROM mytable
TRANSPOSE A, B, C AS VALCOL
  KEY BY KEYCOL
GROUP BY KEYCOL, VALCOL, D;
```

The result table of the TRANSPOSE query is:

KEYCOL	VALCOL	D	COUNT(*)
1	1	d1	1
2	10	d1	1
3	100	d1	1
1	2	d2	1
2	20	d2	1
3	200	d2	1

- This query shows how to use COUNT applied to VALCOL. The result table of the TRANSPOSE query shows the number of distinct values in VALCOL.

```
SELECT COUNT(DISTINCT VALCOL) FROM mytable
TRANSPOSE A, B, C AS VALCOL
  KEY BY KEYCOL
GROUP BY KEYCOL;

(EXPR)
--------------------
                   2
                   2
                   2

--- 3 row(s) selected.
```

- This query shows how multiple TRANSPOSE clauses can be used in the same query. The result table from this query has nine times as many rows as there are rows in MYTABLE:

```
SELECT KEYCOL1, VALCOL1, KEYCOL2, VALCOL2 FROM mytable
TRANSPOSE A, B, C AS VALCOL1
  KEY BY KEYCOL1
TRANSPOSE D, E, F AS VALCOL2
  KEY BY KEYCOL2;
```

The result table of the TRANSPOSE query is:

KEYCOL1	VALCOL1	KEYCOL2	VALCOL2
1	1	1	d1
1	1	2	e1
1	1	3	f1
2	10	1	d1
2	10	2	e1
2	10	3	f1
3	100	1	d1
3	100	2	e1
3	100	3	f1
1	2	1	d2
1	2	2	e2
1	2	3	f2
2	20	1	d2
2	20	2	e2
2	20	3	f2
3	200	1	d2
3	200	2	e2
3	200	3	f2

- This query shows how a TRANSPOSE clause can contain multiple transpose sets—that is, multiple *transpose-item-list* AS *transpose-col-list*. The expressions A, B, and C are of type integer, and expressions D, E, and F are of type character.

```
SELECT KEYCOL, VALCOL1, VALCOL2 FROM mytable
TRANSPOSE A, B, C AS VALCOL1
          D, E, F AS VALCOL2
  KEY BY KEYCOL;
```

The result table of the TRANSPOSE query is:

KEYCOL	VALCOL1	VALCOL2
1	1	?
2	10	?
3	100	?
4	?	d1
5	?	e1
6	?	f1
1	2	?
2	20	?
3	200	?
4	?	d2
5	?	e2
6	?	f2

A question mark (?) in a value column indicates no value for the given KEYCOL.

- This query shows how the preceding query can include a GROUP BY clause:

```
SELECT KEYCOL, VALCOL1, VALCOL2, COUNT(*) FROM mytable
TRANSPOSE A, B, C AS VALCOL1
          D, E, F AS VALCOL2
  KEY BY KEYCOL
GROUP BY KEYCOL, VALCOL1, VALCOL2;
```

The result table of the TRANSPOSE query is:

KEYCOL	VALCOL1	VALCOL2	(EXPR)
1	1	?	1
2	10	?	1
3	100	?	1
1	2	?	1
2	20	?	1
3	200	?	1
4	?	d2	1

KEYCOL	VALCOL1	VALCOL2	(EXPR)
5	?	e2	1
6	?	f2	1
4	?	d1	1
5	?	e1	1
6	?	f1	1

- This query shows how an item in the transpose item list can contain a list of expressions and that the KEY BY clause is optional:

```
SELECT * FROM mytable
TRANSPOSE (1, A, 'abc'), (2, B, 'xyz')
  AS (VALCOL1, VALCOL2, VALCOL3);
```

The result table of the TRANSPOSE query is:

A	B	C	D	E	F	VALCOL1	VALCOL2	VALCOL3
1	10	100	d1	e1	f1	1	1	abc
1	10	100	d1	e1	f1	2	10	xyz
2	20	200	d2	e2	f2	1	2	abc
2	20	200	d2	e2	f2	2	20	xyz

8 SQL/MX Functions and Expressions

This section describes the syntax and semantics of specific functions and expressions that you can use in NonStop SQL/MX statements. The functions and expressions are categorized according to their functionality.

Categories

Use these types of functions within an SQL value expression:

For more information on SQL value expressions, see Expressions on page 6-41.

Table-valued stored functions cannot be used within an SQL value expression. See Table-Valued Stored Functions on page 8-9.

Aggregate (Set) Functions

An aggregate (or set) function operates on a group or groups of rows retrieved by the SELECT statement or the subquery in which the aggregate function appears.

AVG Function on page 8-14	Computes the average of a group of numbers derived from the evaluation of the expression argument of the function.
COUNT Function on page 8-38	Counts the number of rows that result from a query (by using *) or the number of rows that contain a distinct value in the one-column table derived from the expression argument of the function (optionally distinct values).
MAX Function on page 8-103	Determines a maximum value from the group of values derived from the evaluation of the expression argument.
MIN Function on page 8-104	Determines a minimum value from the group of values derived from the evaluation of the expression argument.
STDDEV Function on page 8-171	Computes the statistical standard deviation of a group of numbers derived from the evaluation of the expression argument of the function. The numbers can be weighted.
SUM Function on page 8-176	Computes the sum of a group of numbers derived from the evaluation of the expression argument of the function.
VARIANCE Function on page 8-207	Computes the statistical variance of a group of numbers derived from the evaluation of the expression argument of the function. The numbers can be weighted.

Note that columns and expressions can be arguments of an aggregate function. The expressions cannot contain aggregate functions or subqueries.

An aggregate function can accept an argument specified as DISTINCT, which eliminates duplicate values before the aggregate function is applied. Only one DISTINCT aggregate function is allowed at each level of a SELECT statement. Multiple DISTINCT aggregates are allowed if they are on the same column but are not permitted on different columns. Exceptions to this rule include MIN and MAX functions and aggregate functions with unique columns or expressions for which DISTINCT is unnecessary. See DISTINCT Aggregate Functions on page 2-349.

If you include a GROUP BY clause in the SELECT statement, the columns you refer to in the select list must be either grouping columns or arguments of an aggregate function. If you do not include a GROUP BY clause but you specify an aggregate function in the select list, all rows of the SELECT result table form the one and only group.

See the individual entry for the function.

Character String Functions

These functions manipulate character strings. These functions either use a character value expression as an argument or return a result of character data type:

ASCII Function on page 8-11	Returns the ASCII code value of the first character of a ISO88591 character value expression.
CHAR Function on page 8-23	Returns the specified code value in a character set.
CHAR_LENGTH Function on page 8-24	Returns the number of characters in a string. You can also use CHARACTER_LENGTH.
CONCAT Function on page 8-34	Returns the concatenation of two character value expressions as a string value. You can also use the concatenation operator (||).
INSERT Function on page 8-84	Returns a character string where a specified number of characters within the character string have been deleted and then a second character string has been inserted at a specified start position.
LCASE Function on page 8-87	Downshifts characters. You can also use LOWER.
LEFT Function on page 8-88	Returns the leftmost specified number of characters from a character expression.
LOWER Function on page 8-94	Downshifts single-byte characters. You can also use LCASE.
LPAD Function on page 8-99	Replaces the leftmost specified number of characters in a character expression with a padding character.
LTRIM Function on page 8-102	Removes the specified characters from the left of the character string.

POSITION Function on page 8-131	Returns the position of a specified substring within a character string. You can also use LOCATE.
REPEAT Function on page 8-145	Returns a character string composed of the evaluation of a character expression repeated a specified number of times.
REPLACE Function on page 8-146	Returns a character string where all occurrences of a specified character string in the original string are replaced with another character string.
RIGHT Function on page 8-147	Returns the rightmost specified number of characters from a character expression.
RPAD Function on page 8-150	Replaces the rightmost specified number of characters in a character expression with a padding character.
RTRIM Function on page 8-152	Removes the specified characters from the right of the character string.
SPACE Function on page 8-170	Returns a character string consisting of a specified number of spaces.
SUBSTRING Function on page 8-174	Extracts a substring from a character string.
TO_CHAR(<NUMERIC>) Function on page 8-181	Converts a numeric data to VARCHAR as per the specified format.
TO_CHAR(<DATETIME>) Function on page 8-185	Converts datetime data to VARCHAR as per the specified format.
TRIM Function on page 8-192	Removes the specified characters from the left side, the right side, or both sides of a character string.
UCASE Function on page 8-193	Upshifts single-byte characters. You can also use UPSHIFT or UPPER.
UPPER Function on page 8-201	Upshifts single-byte characters. You can also use UPSHIFT or UCASE.
UPSHIFT Function on page 8-202	Upshifts single-byte characters. You can also use UPPER or UCASE.

See the individual entry for the function.

Datetime Functions

These functions use either a datetime value expression as an argument or return a result of datetime data type:

HOUR Function on page 8-83	Returns an integer value in the range 0 through 23 that represents the corresponding hour of the day.
JULIANTIMESTAMP Function on page 8-85	Converts a datetime value to a Julian timestamp.
MINUTE Function on page 8-105	Returns an integer value in the range 0 through 59 that represents the corresponding minute of the hour.
MONTH Function on page 8-107	Returns an integer value in the range 1 through 12 that represents the corresponding month of the year.
MONTHNAME Function on page 8-108	Returns a character literal that is the name of the month of the year (January, February, and so on).
QUARTER Function on page 8-133	Returns an integer value in the range 1 through 4 that represents the corresponding quarter of the year.
SECOND Function on page 8-167	Returns an integer value in the range 0 through 59 that represents the corresponding second of the minute.
WEEK Function on page 8-212	Returns an integer value in the range 1 through 54 that represents the corresponding week of the year.
YEAR Function on page 8-213	Returns an integer value that represents the year.

See the individual entry for the function.

Mathematical Functions

Use these mathematical functions within an SQL numeric value expression:

ABS Function on page 8-10	Returns the absolute value of a numeric value expression.
ACOS Function on page 8-10	Returns the arccosine of a numeric value expression as an angle expressed in radians.
ASIN Function on page 8-12	Returns the arcsine of a numeric value expression as an angle expressed in radians.
ATAN Function on page 8-13	Returns the arctangent of a numeric value expression as an angle expressed in radians.
ATAN2 Function on page 8-13	Returns the arctangent of the x and y coordinates, specified by two numeric value expressions, as an angle expressed in radians.
CEILING Function on page 8-22	Returns the smallest integer greater than or equal to a numeric value expression.

COS Function on page 8-37	Returns the cosine of a numeric value expression, where the expression is an angle expressed in radians.
COSH Function on page 8-37	Returns the hyperbolic cosine of a numeric value expression, where the expression is an angle expressed in radians.
EXP Function on page 8-65	Returns the exponential value (to the base e) of a numeric value expression.
FLOOR Function on page 8-78	Returns the largest integer less than or equal to a numeric value expression.
LOG Function on page 8-93	Returns the natural logarithm of a numeric value expression.
LOG10 Function on page 8-93	Returns the base 10 logarithm of a numeric value expression.
MOD Function on page 8-106	Returns the remainder (modulus) of an integer value expression divided by an integer value expression.
PI Function on page 8-130	Returns the constant value of pi as a floating-point value.
POWER Function on page 8-132	Returns the value of a numeric value expression raised to the power of an integer value expression. You can also use the exponential operator **.
RADIANS Function on page 8-143	Converts a numeric value expression expressed in degrees to the number of radians.
SIGN Function on page 8-168	Returns an indicator of the sign of a numeric value expression. If value is less than zero, returns -1 as the indicator. If value is zero, returns 0. If value is greater than zero, returns 1.
SIN Function on page 8-169	Returns the sine of a numeric value expression, where the expression is an angle expressed in radians.
SINH Function on page 8-169	Returns the hyperbolic sine of a numeric value expression, where the expression is an angle expressed in radians.
SQRT Function on page 8-170	Returns the square root of a numeric value expression.
TAN Function on page 8-178	Returns the tangent of a numeric value expression, where the expression is an angle expressed in radians.
TANH Function on page 8-178	Returns the hyperbolic tangent of a numeric value expression, where the expression is an angle expressed in radians.

See the individual entry for the function.

Sequence Functions

Sequence functions operate on ordered rows of the intermediate result table of a SELECT statement that includes a SEQUENCE BY clause. Sequence functions are categorized generally as difference, moving, offset, or running.

Difference sequence functions:

DIFF1 Function on page 8-59	Calculates differences between values of a column expression in the current row and previous rows.
DIFF2 Function on page 8-62	Calculates differences between values of the result of DIFF1of the current row and DIFF1 of previous rows.

Moving sequence functions:

MOVINGAVG Function on page 8-109	Returns the average of non-null values of a column expression in the current window.
MOVINGCOUNT Function on page 8-111	Returns the number of non-null values of a column expression in the current window.
MOVINGMAX Function on page 8-113	Returns the maximum of non-null values of a column expression in the current window.
MOVINGMIN Function on page 8-115	Returns the minimum of non-null values of a column expression in the current window.
MOVINGSTDDEV Function on page 8-117	Returns the standard deviation of non-null values of a column expression in the current window.
MOVINGSUM Function on page 8-119	Returns the sum of non-null values of a column expression in the current window.
MOVINGVARIANCE Function on page 8-121	Returns the variance of non-null values of a column expression in the current window.

Offset sequence function:

OFFSET Function on page 8-128	Retrieves columns from previous rows.

Running sequence functions:

RUNNINGAVG Function on page 8-153	Returns the average of non-null values of a column expression up to and including the current row.
RUNNINGCOUNT Function on page 8-155	Returns the number of rows up to and including the current row.
RUNNINGMAX Function on page 8-157	Returns the maximum of values of a column expression up to and including the current row.
RUNNINGMIN Function on page 8-159	Returns the minimum of values of a column expression up to and including the current row.
RUNNINGSTDDEV Function on page 8-161	Returns the standard deviation of non-null values of a column expression up to and including the current row.

RUNNINGSUM Function on page 8-163	Returns the sum of non-null values of a column expression up to and including the current row.
RUNNINGVARIANCE Function on page 8-165	Returns the variance of non-null values of a column expression up to and including the current row.

Other sequence functions:

LASTNOTNULL Function on page 8-86	Returns the last non-null value for the specified column expression. If only null values have been returned, returns null.
ROWS SINCE Function on page 8-148	Returns the number of rows counted since the specified condition was last true.
THIS Function on page 8-179	Used in ROWS SINCE to distinguish between the value of the column in the current row and the value of the column in previous rows.

See SEQUENCE BY Clause on page 7-19 and the individual entry for each function.

Other Functions and Expressions

Use these other functions and expressions in an SQL value expression:

CASE (Conditional) Expression on page 8-16	A conditional expression. The two forms of the CASE expression are simple and searched.
CAST Expression on page 8-20	Converts a value from one data type to another data type that you specify.
CURRENT_USER Function on page 8-43	Returns the Guardian user name corresponding to the current authorization ID. This function is equivalent to SESSION_USER and USER.
HASHPARTFUNC Function on page 8-79	Returns the number of the partition to which a specified partitioning key belongs
SESSION_USER Function on page 8-168	Returns the Guardian user name corresponding to the current authorization ID. This function is equivalent to CURRENT_USER and USER.
USER Function on page 8-203	Returns the Guardian user name corresponding to the current authorization ID. This function is equivalent to CURRENT_USER and SESSION_USER.
NVL Function on page 8-123	Returns a specified value when the expression is NULL.
NVL2 Function on page 8-125	Returns a specified value when the expression is NULL and, if the expression is not NULL, returns another specified value.

LNNVL Function on page 8-89	Returns TRUE if the condition is FALSE or NULL, and FALSE when the condition is TRUE.
DECODE Function on page 8-55	Compares the given expression to a set of specified conditions one by one in the specified order. Returns the value corresponding to the matching condition.
COALESCE Function on page 8-27	Returns the value of the first expression in the list that is not NULL. If all the expressions in the list result in NULL, the function returns NULL.

See the individual entry for the function.

Table-Valued Stored Functions

Table-valued stored functions are system-defined functions that generate a result table. Use these functions anywhere a table reference can be used in a SELECT statement or the SELECT form of a subquery:

COMPILERCONTROLS Function on page 8-31	Retrieves the active control settings, such as, CQDs, CTs, CQS from the compiler.
EXPLAIN Function on page 8-66	Builds a result table that describes the access plan of a DML statement, which can then be queried by a SELECT statement.
FEATURE_VERSION_INFO Function on page 8-76	Returns feature version information for all user objects with an object feature version (OFV) higher than a given value, in a specified set of catalogs.
QUERYCACHE Function on page 8-134	Collects and returns the current state of the query plan cache statistics in a single-row result table.
QUERYCACHEENTRIES Function on page 8-138	Collects and returns the query plan cache statistics in a result table with one row for each entry of the cache.
RELATEDNESS Function on page 8-144	Returns relatedness information for a single entity.
VERSION_INFO Function on page 8-204	Returns version information for a single entity.

See the individual entry for the function.

ABS Function

The ABS function returns the absolute value of a numeric value expression.

ABS is an SQL/MX extension.

```
ABS (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the ABS function. The result is returned as an unsigned numeric value if the precision of the argument is less than 10 or as a LARGEINT if the precision of the argument is greater than or equal to 10. See Numeric Value Expressions on page 6-52.

Examples of ABS

- This function returns the value 8:

  ```
  ABS (-20 + 12)
  ```

ACOS Function

The ACOS function returns the arccosine of a numeric value expression as an angle expressed in radians.

ACOS is an SQL/MX extension.

```
ACOS (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the ACOS function. The range for the value of the argument is from -1 to +1. See Numeric Value Expressions on page 6-52.

Examples of ACOS

- This function returns the value 3.49044274380724352E-001 or approximately 0.3491 in radians (which is 20 degrees):

  ```
  ACOS (0.9397)
  ```

- This function returns the value 0.3491. The function ACOS is the inverse of the function COS.

  ```
  ACOS (COS (0.3491))
  ```

ASCII Function

The ASCII function returns the integer that is the ASCII code of the first character in a character string expression associated with the ISO8891 character set.

ASCII is an SQL/MX extension.

```
ASCII (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters. See Character Value Expressions on page 6-41.

Examples of ASCII

- Select the column JOBDESC and return the ASCII code of the first character of the job description:

```
SELECT jobdesc, ASCII (jobdesc)
FROM persnl.job;

JOBDESC              (EXPR)
-----------------    --------
MANAGER              77
PRODUCTION SUPV      80
ASSEMBLER            65
SALESREP             83
...                  ...

--- 10 row(s) selected.
```

ASIN Function

The ASIN function returns the arcsine of a numeric value expression as an angle expressed in radians.

ASIN is an SQL/MX extension.

```
ASIN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the ASIN function. The range for the value of the argument is from -1 to +1. See Numeric Value Expressions on page 6-52.

Examples of ASIN

- This function returns the value 3.49044414403046464E-001 or approximately 0.3491 in radians (which is 20 degrees):

  ```
  ASIN (0.3420)
  ```

- This function returns the value 0.3491. The function ASIN is the inverse of the function SIN.

  ```
  ASIN (SIN (0.3491))
  ```

ATAN Function

The ATAN function returns the arctangent of a numeric value expression as an angle expressed in radians.

ATAN is an SQL/MX extension.

```
ATAN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the ATAN function. See Numeric Value Expressions on page 6-52.

Examples of ATAN

- This function returns the value 8.72766423249958400E-001 or approximately 0.8727 in radians (which is 50 degrees):

  ```
  ATAN (1.192)
  ```

- This function returns the value 0.8727. The function ATAN is the inverse of the function TAN.

  ```
  ATAN (TAN (0.8727))
  ```

ATAN2 Function

The ATAN2 function returns the arctangent of the x and y coordinates, specified by two numeric value expressions, as an angle expressed in radians.

ATAN2 is an SQL/MX extension.

```
ATAN2 (numeric-expression-x,numeric-expression-y)
```

numeric-expression-x,numeric-expression-y

are SQL numeric value expressions that specify the value for the x and y coordinate arguments of the ATAN2 function. See Numeric Value Expressions on page 6-52.

Examples of ATAN2

- This function returns the value 2.66344329881899600E+000, or approximately 2.6634:

  ```
  ATAN2 (1.192,-2.3)
  ```

AVG Function

AVG is an aggregate function that returns the average of a set of numbers.

```
AVG ([ALL | DISTINCT] expression)
```

`ALL | DISTINCT`

specifies whether duplicate values are included in the computation of the AVG of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the AVG function is applied.

expression

specifies a numeric or interval value *expression* that determines the values to average. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the AVG function operates on distinct values from the one-column table derived from the evaluation of *expression*.

See Numeric Value Expressions on page 6-52 and Interval Value Expressions on page 6-47.

Considerations for AVG

Data Type of the Result

The data type of the result depends on the data type of the argument. If the argument is an exact numeric type, the result is LARGEINT. If the argument is an approximate numeric type, the result is DOUBLE PRECISION. If the argument is INTERVAL data type, the result is INTERVAL with the same precision as the argument.

The scale of the result is the same as the scale of the argument. If the argument has no scale, the result is truncated.

Operands of the Expression

The expression includes columns from the rows of the SELECT result table but cannot include an aggregate function. These expressions are valid:

```
AVG (SALARY)
AVG (SALARY * 1.1)
AVG (PARTCOST * QTY_ORDERED)
```

Nulls

All nulls are eliminated before the function is applied to the set of values. If the result table is empty, AVG returns NULL.

Examples of AVG

- Return the average value of the SALARY column:

```
SELECT AVG (salary)
FROM persnl.employee;

(EXPR)
---------------------
             49441.52

--- 1 row(s) selected.
```

- Return the average value of the set of unique SALARY values:

```
SELECT AVG(DISTINCT salary) AS Avg_Distinct_Salary
FROM persnl.employee;

AVG_DISTINCT_SALARY
---------------------
             53609.89

--- 1 row(s) selected.
```

- Return the average salary by department:

```
SELECT deptnum, AVG (salary) AS "AVERAGE SALARY"
FROM persnl.employee
WHERE deptnum < 3000
GROUP BY deptnum;

Dept/Num  "AVERAGE SALARY"
--------  ---------------------
    1000               52000.17
    2000               50000.10
    1500               41250.00
    2500               37000.00

--- 4 row(s) selected.
```

CASE (Conditional) Expression

Considerations for CASE
Examples of CASE

The CASE expression is a conditional expression with two forms: simple and searched.

In a simple CASE expression, NonStop SQL/MX compares a value to a sequence of values and sets the CASE expression to the value associated with the first match—if there is a match. If there is no match, NonStop SQL/MX returns the value specified in the ELSE clause (which can be null).

In a searched CASE expression, NonStop SQL/MX evaluates a sequence of conditions and sets the CASE expression to the value associated with the first condition that is true—if there is a true condition. If there is no true condition, NonStop SQL/MX returns the value specified in the ELSE clause (which can be null).

```
Simple CASE is:

CASE case-expression
  WHEN expression-1 THEN {result-expression-1 | NULL}
  WHEN expression-2 THEN {result-expression-2 | NULL}
  ...
  WHEN expression-n THEN {result-expression-n | NULL}
  [ELSE {result-expression | NULL}]
END

Searched CASE is:

CASE
  WHEN condition-1 THEN {result-expression-1 | NULL}
  WHEN condition-2 THEN {result-expression-2 | NULL}
  ...
  WHEN condition-n THEN {result-expression-n | NULL}
  [ELSE {result-expression | NULL}]
END
```

case-expression

specifies a value expression that is compared to the value expressions in each WHEN clause of a simple CASE. The data type of each *expression* in the WHEN clause must be comparable to the data type of *case-expression*.

expression-1 ... expression-n

specifies a value associated with each *result-expression*. If the value of an *expression* in a WHEN clause matches the value of *case-expression*, simple CASE returns the associated *result-expression* value. If there is no match, the CASE expression returns the value expression specified in the ELSE clause, or NULL if the ELSE value is not specified.

result-expression-1 ... *result-expression-n*

specifies the result value expression associated with each *expression* in a WHEN clause of a simple CASE, or with each *condition* in a WHEN clause of a searched CASE. All of the *result-expressions* must have comparable data types, and at least one of the *result-expressions* must return non-null.

result-expression

follows the ELSE keyword and specifies the value returned if none of the expressions in the WHEN clause of a simple CASE are equal to the case expression, or if none of the conditions in the WHEN clause of a searched CASE are true. If the ELSE *result-expression* clause is not specified, CASE returns NULL. The data type of *result-expression* must be comparable to the other results.

condition-1 ... *condition-n*

specifies conditions to test for in a searched CASE. If a *condition* is true, the CASE expression returns the associated *result-expression* value. If no *condition* is true, the CASE expression returns the value expression specified in the ELSE clause, or NULL if the ELSE value is not specified.

Considerations for CASE

Data Type of the CASE Expression

The data type of the result of the CASE expression depends on the data types of the result expressions. If the results all have the same data type, the CASE expression adopts that data type. If the results have comparable but not identical data types, the CASE expression adopts the data type of the union of the result expressions. This result data type is determined in these ways.

Character Data Type

If any data type of the result expressions is variable-length character string, the result data type is variable-length character string with maximum length equal to the maximum length of the result expressions.

If any one of the result expressions is a character string constant, the result data type is a variable-length character string with a maximum length equal to the maximum length of the result expressions.

However, if either of the data types is not a variable-length character string and the result expressions is not a character string constant, the result data type is fixed-length character string with a length equal to the maximum of the lengths of the result expressions.

Numeric Data Type

If all of the data types of the result expressions are exact numeric, the result data type is exact numeric with precision and scale equal to the maximum of the precisions and scales of the result expressions.

For example, if *result-expression-1* and *result-expression-2* have data type NUMERIC(5) and *result-expression-3* has data type NUMERIC(8,5), the result data type is NUMERIC(10,5).

If any data type of the result expressions is approximate numeric, the result data type is approximate numeric with precision equal to the maximum of the precisions of the result expressions.

Datetime Data Type

If the data type of the result expressions is datetime, the result data type is the same datetime data type.

Interval Data Type

If the data type of the result expressions is interval, the result data type is the same interval data type (either year-month or day-time) with the start field being the most significant of the start fields of the result expressions and the end field being the least significant of the end fields of the result expressions.

Examples of CASE

- Use a simple CASE to decode JOBCODE and return NULL if JOBCODE does not match any of the listed values:

```
SELECT last_name, first_name,
  CASE jobcode
    WHEN 100 THEN 'MANAGER'
    WHEN 200 THEN 'PRODUCTION SUPV'
    WHEN 250 THEN 'ASSEMBLER'
    WHEN 300 THEN 'SALESREP'
    WHEN 400 THEN 'SYSTEM ANALYST'
    WHEN 420 THEN 'ENGINEER'
    WHEN 450 THEN 'PROGRAMMER'
    WHEN 500 THEN 'ACCOUNTANT'
    WHEN 600 THEN 'ADMINISTRATOR ANALYST'
    WHEN 900 THEN 'SECRETARY'
    ELSE NULL
  END
FROM persnl.employee;

LAST_NAME             FIRST_NAME       (EXPR)
--------------------  ---------------  -----------------

GREEN                 ROGER            MANAGER
HOWARD                JERRY            MANAGER
```

```
RAYMOND                    JANE              MANAGER
...
CHOU                       JOHN              SECRETARY
CONRAD                     MANFRED           PROGRAMMER
HERMAN                     JIM               SALESREP
CLARK                      LARRY             ACCOUNTANT
HALL                       KATHRYN           SYSTEM ANALYST
...

--- 62 row(s) selected.
```

- Use a searched CASE to return LAST_NAME, FIRST_NAME and a value based on SALARY that depends on the value of DEPTNUM:

```
SELECT last_name, first_name, deptnum,
  CASE
    WHEN deptnum = 9000 THEN salary * 1.10
    WHEN deptnum = 1000 THEN salary * 1.12
    ELSE salary
  END
FROM persnl.employee;

LAST_NAME         FIRST_NAME    DEPTNUM  (EXPR)
---------------- ------------  -------  -------------------
GREEN             ROGER            9000         193050.0000
HOWARD            JERRY            1000         153440.1120
RAYMOND           JANE             3000         136000.0000
...
--- 62 row(s) selected.
```

CAST Expression

Considerations for CAST
Valid Conversions for CAST
Examples of CAST

The CAST expression converts data to the data type you specify.

```
CAST ({expression | NULL} AS data-type)
```

expression | NULL

specifies the operand to convert to the data type *data-type*.

If the operand is an *expression*, then *data-type* depends on the data type of *expression* and follows the rules outlined in Valid Conversions for CAST.

If the operand is NULL, or if the value of the *expression* is null, the result of CAST is NULL, regardless of the data type you specify.

data-type

specifies a data type to associate with the operand of CAST. See Data Types on page 6-17.

When casting data to a CHAR or VARCHAR data type, the resulting data value is left justified. Otherwise, the resulting data value is right justified. Further, when you are casting to a CHAR or VARCHAR data type, you must specify the length of the target value.

Considerations for CAST

Depending on how your file is set up, using CAST might cause poor query performance by preventing the optimizer from choosing the most efficient plan and requiring the executor to perform a complete table or index scan.

Valid Conversions for CAST

- An exact or approximate numeric value to any other numeric data type. The size of the character string should be large enough to hold the numeric value without truncation. An error 8402 is returned if the size of the character string cannot hold the numeric value without truncation.
- An exact or approximate numeric value to any character string data type.
- An exact numeric value to either a single-field year-month or day-time interval such as INTERVAL ‘30’ DAY.
- A character string to any other data type, with one restriction:

The contents of the character string to be converted must be consistent in meaning with the data type of the result. For example, if you are converting to DATE, the contents of the character string must be 10 characters consisting of the year, a hyphen, the month, another hyphen, and the day.

- A date value to a character string or to a TIMESTAMP (NonStop SQL/MX fills in the time part with 00:00:00.00).
- A time value to a character string or to a TIMESTAMP (NonStop SQL/MX fills in the date part with the current date).
- A timestamp value to a character string, a DATE, a TIME, or another TIMESTAMP with different fractional seconds precision.
- A year-month interval value to a character string, an exact numeric, or to another year-month INTERVAL with a different start field precision.
- A day-time interval value to a character string, an exact numeric, or to another day-time INTERVAL with a different start field precision.

Examples of CAST

- This example returns the difference of two timestamps in minutes:

```
CAST((d.step_end - d.step_start) AS INTERVAL MINUTE)
```

- The PROJECT table contains a column START_DATE of data type DATE and a column SHIP_TIMESTAMP of data type TIMESTAMP.

  Use CAST to return the number of days for completion of a project:

```
SELECT projdesc, start_date, ship_timestamp,
  (CAST (ship_timestamp AS DATE) - start_date) DAY
FROM persnl.project;

PROJDESC       START_DATE SHIP_TIMESTAMP             (EXPR)
-------------- ---------- -------------------------- ------
SALT LAKE CITY 1996-04-10 1996-04-21 08:15:00.000000     11
ROSS PRODUCTS  1996-06-10 1996-07-21 08:30:00.000000     41
MONTANA TOOLS  1996-10-10 1996-12-21 09:00:00.000000     72
AHAUS TOOL     1996-08-21 1996-10-21 08:10:00.000000     61
THE WORKS      1996-09-21 1996-10-21 10:15:00.000000     30

--- 5 row(s) selected.
```

  Note that DATE differences can be expressed only in the number of days, the least significant unit of measure for dates. (An interval is either year-month or day-time.) In this example, the result is the same if you express the difference as:

```
CAST (ship_timestamp AS DATE) - start_date
```

  You are not required to specify the interval qualifier.

- Suppose that your database includes a log file of user information. This example converts the current timestamp to a character string and concatenates the result to a character literal. Note the length must be specified.

```
INSERT INTO stats.logfile
(user_key, user_info)
VALUES (001, 'User JBrook, executed at ' ||
        CAST (CURRENT_TIMESTAMP AS CHAR(26)));
```

CEILING Function

The CEILING function returns the smallest integer, represented as a FLOAT data type, greater than or equal to a numeric value expression.

CEILING is an SQL/MX extension.

```
CEILING (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the CEILING function. See Numeric Value Expressions on page 6-52.

Examples of CEILING

- This function returns the integer value 3.00000000000000064E+000, represented as a FLOAT data type:

```
CEILING (2.25)
```

CHAR Function

The CHAR function returns the character that has the specified code value, which must be of exact numeric with scale 0.

CHAR is an SQL/MX extension.

```
CHAR(code-value, [,char-set-name])
```

code-value

is a valid code value in the character set in use.

char-set-name

can be ISO88591, KANJI, KSC5601, or UCS2. The returned character will be associated with the character set specified by *char-set-name* with the DEFAULT collation.

The default is ISO88591.

Examples of CHAR

- Select the column CUSTNAME and return the ASCII code of the first character of the customer name and its CHAR value:

```
SELECT custname, ASCII (custname), CHAR (ASCII (custname))
FROM sales.customer;

CUSTNAME            (EXPR)   (EXPR)
------------------  -------  -------
CENTRAL UNIVERSITY  67       C
BROWN MEDICAL CO    66       B
STEVENS SUPPLY      83       S
PREMIER INSURANCE   80       P
...                 ...      ...

--- 15 row(s) selected.
```

CHAR_LENGTH Function

Considerations for CHAR_LENGTH
SQL/MP Considerations for CHAR_LENGTH
Examples of CHAR_LENGTH

The CHAR_LENGTH function returns the number of characters in a string. You can also use CHARACTER_LENGTH.

```
CHAR[ACTER]_LENGTH (string-value-expression)
```

string-value-expression

specifies the string value expression for which to return the length in characters. NonStop SQL/MX returns the result as a two-byte signed integer with a scale of zero. If *string-value-expression* is null, NonStop SQL/MX returns a length of null. See Character Value Expressions on page 6-41.

Considerations for CHAR_LENGTH

CHAR and VARCHAR Operands

For a column declared as fixed CHAR, NonStop SQL/MX returns the maximum length of that column. For a VARCHAR column, NonStop SQL/MX returns the actual length of the string stored in that column.

SQL/MP Considerations for CHAR_LENGTH

Similarity to OCTET_LENGTH Function

The CHAR_LENGTH and OCTET_LENGTH functions are similar. The CHAR_LENGTH function returns the number of characters in a string, and the OCTET_LENGTH function returns the number of bytes in a string.

For example, suppose that an SQL/MP table has been created in this way:

```
CREATE TABLE tab (col_kanji CHAR(10) CHARACTER SET KANJI,
                  col_char CHAR(10));
```

This row is inserted into the table:

```
INSERT INTO tab VALUES (_KANJI'kkkk', 'ccc');
```

This SELECT statement returns the same values for the character length and the octet length of the ISO88591 column. One character of an ISO88591 character set is equivalent to one byte.

```
SELECT CHAR_LENGTH(col_char) AS CHARLENGTH_CHAR,
       OCTET_LENGTH(col_char) AS OCTETLENGTH_CHAR
FROM tab;

CHARLENGTH_CHAR  OCTETLENGTH_CHAR
---------------  ----------------

             10                10

--- 1 row(s) selected.
```

This SELECT statement returns the same values for the character length and the octet length of the KANJI column.

```
SELECT CHAR_LENGTH(col_kanji) AS CHARLENTH_KANJI,
       OCTET_LENGTH(col_kanji) AS OCTETLENGTH_KANJI
FROM tab;

CHARLENTH_KANJI  OCTETLENGTH_KANJI
---------------  -----------------
             10                 20

--- 1 row(s) selected.
```

Examples of CHAR_LENGTH

- This function returns 12 as the result. The concatenation operator is denoted by two vertical bars (||).

  ```
  CHAR_LENGTH ('ROBERT' || ' ' || 'SMITH')
  ```

- The string `''` is the null (or empty) string. This function returns 0 (zero):

  ```
  CHAR_LENGTH ('')
  ```

- The DEPTNAME column has data type CHAR(12). Therefore, this function always returns 12:

  ```
  CHAR_LENGTH (deptname)
  ```

- The PROJDESC column in the PROJECT table has data type VARCHAR(18). This function returns the actual length of the column value—not 18 for shorter strings—because it is a VARCHAR value:

```
SELECT CHAR_LENGTH (projdesc)
FROM persnl.project;

(EXPR)
----------

        14
        13
        13
        17
         9
         9
--- 6 row(s) selected.
```

COALESCE Function

Returns the value of the first expression in the list that is NOT NULL or if all the expressions in the list result in NULL, the function returns NULL.

```
COALESCE (expr1, expr2[,expr3 ...])
```

expr1, *expr2* , *expr3* and so on are SQL value expressions of comparable data types.

Considerations

- If all the expressions in the argument list are fixed-length character types, the return value is a fixed-length character string equal to the maximum size of all the expression value types.
- If any of the expressions in the argument list are variable-length character type, the return value is a variable-length character string with the maximum size of all the expression value types.
- If all the expressions in the argument list are integer types, the return value is of the same data type as the largest integer type of all the possible return values.
- If all the expressions in the argument list are numeric types, and at least one is REAL, FLOAT, or DOUBLE PRECISION, the return value is a DOUBLE PRECISION type.
- If all the expressions in the argument list are numeric types, none are REAL, FLOAT, or DOUBLE PRECISION, and at least one is of type NUMERIC, the return value is a NUMERIC type.
- The last expression in the COALESCE function cannot be NULL. This is a NonStop SQL/MX restriction.
- If the return value is of type NUMERIC or DECIMAL, the value has a precision value equal to the sum of:
    - The maximum scale of all the return values.
    - The maximum of precision minus scale value for all the return values.

    However, the precision value must not exceed 18.
- If the return value is of type NUMERIC or DECIMAL, the scale of the return value is the minimum of:
    - The maximum scale of all the return values.
    - 18 - (the maximum of precision minus scale value for all the return values).
- The COALESCE function is internally transformed into the SQL/MX CASE…WHEN…ELSE…END function, and therefore all typing rules that apply to the CASE…WHEN…ELSE…END also apply to this function.

- The COALESCE function has no limit on the number of arguments, other than the general limit of an SQL expression. However, large lists of expressions do not perform well.

Note. Existing NonStop SQL/MX conversions rules are applied to the above rules**.**

Examples of COALESCE

- This function returns the first NOT NULL value from the expression list (msg, warn, err). For the first row, value for column `warn` is returned. For the second row, value for column `err` is returned.

```
create table tab1(msg varchar(20), warn varchar(20), err
varchar(20));

-- The Default for column msg, column warn and column err is
NULL
insert into tab1(warn) values('xxxx');
insert into tab1(err) values('zzzz');

-- In the first row
select COALESCE(msg,warn,err), msg, warn, err from tab1;

>>select COALESCE(msg,warn,err), msg, warn, err from tab1;

(EXPR)        MSG           WARN          ERR

----         -------       -------       -----

xxxx          ?             xxxx          ?

zzzz          ?             ?             zzzz
```

- The function returns the language marks for the students. Assume that each student can only choose one language course (English, French, Spanish, or German).

```
Create table student( name varchar(20),
math int,
science int,
economics int,
english int,
french int,
spanish int,
german int);
--- SQL operation complete.


-- This inserts the French marks for John, in addition to
math, science and economics.
insert into student(name, math, science, economics, french)
```

```
values('John', 60, 70, 80, 60);
--- 1 row(s) inserted.

--  This inserts the English marks for Harry, in addition to
math, science and economics.
insert into student(name, math, science, economics, english)
values('Harry', 50, 80, 75, 55);
--- 1 row(s) inserted.

--  This inserts the German marks for Mike, in addition to
math, science and economics.
insert into student(name, math, science, economics, german)
values('Mike', 90, 50, 60, 80);
--- 1 row(s) inserted.

--  This inserts the Spanish marks for Raul, in addition to
math, science and economics.
insert into student(name, math, science, economics, spanish)
values('Raul', 60, 70, 80, 70);
--- 1 row(s) inserted.

--  This inserts the Spanish marks for Leo, in addition to
math, science and economics.
insert into student(name, math, science, economics, spanish)
values('Leo', 60, 70, 80, 35);
--- 1 row(s) inserted.

--  This inserts the English marks for Donald, in addition to
math, science and economics.
insert into student(name, math, science, economics, english)
values('Donald',60, 70, 80, 25);
--- 1 row(s) inserted.


-- The COALESCE function selects the language marks opted by
each student.
select name, math, science, economics, coalesce(english,
french, spanish, german) as language_mark from student;

NAME      MATH          SCIENCE      ECONOMICS   LANGUAGE_MARK

--------------------  -----------  -----------  -----------

John       60            70            80              60

Harry      50            80            75              55

Mike       90            50            60              80

Raul       60            70            80              70

Leo        60            70            80              35

Donald     60            70            80              25

--- 6 row(s) selected.
```

CODE_VALUE Function

The CODE_VALUE function returns an unsigned integer (INTEGER UNSIGNED) that is the code point of the first character in a character value expression that can be associated with any character sets allowed.

CODE_VALUE is an SQL/MX extension.

```
CODE_VALUE(character-value-expression)
```

character-value-expression

is a character string.

Considerations for CODE_VALUE Function

- This function returns 97 as the result:

```
>>select code_value('abc') from (values(1))x;

(EXPR)
----------
        97
```

COMPILERCONTROLS Function

Considerations for COMPILERCONTROLS

Examples of COMPILERCONTROLS

The COMPILERCONTROLS function is an SQL/MX extension.

The COMPILERCONTROLS function can be specified as a table reference (*table*) in the FROM clause of a SELECT statement if it is preceded by the keyword TABLE and surrounded by parentheses. The syntax for the COMPILERCONTROLS function has no parameters.

```
COMPILERCONTROLS ()
```

In a dynamic environment (that is, MXCI, MXCS, JDBC, or dynamic SQL), the COMPILERCONTROLS function returns the active control query defaults (CQDs) and CTDs from the SQL/MX compiler.

Considerations for COMPILERCONTROLS

Using SELECT and COMPILERCONTROLS

The SELECT statement displays the selected columns from the COMPILERCONTROLS function:

Column Name	Type/Size	Description
SeqNum	INTEGER	Organizes the attribute values that are larger than 78 characters.
Type	CHAR(4)	Indicates the type of the attribute being displayed. Supported values are CQD, CT, CS, and CQS.
State	CHAR(8)	Indicates how the attribute is set. Supported values are: DEF_TAB, BY_SYS, DEFAULT, BY_USER, NOT_SET, RD_ONLY.
Attribute	CHAR(50)	Displays the name of the attribute.
Attribute_Value	CHAR(78)	Provides the value of the attribute. This column can have actual values larger than 78 characters, where the values will be split into chunks of 78 characters and displayed. The SeqNum column is used to organize the chunks.
Table_Name	CHAR(386)	Populates only for the Control table type. Only the first 78 characters of the table name are displayed.

Examples of COMPILERCONTROLS

- To display control tables, set the following CTs to ON:

```
>>control table t3 mdam 'on';
--- SQL operation complete.

>>control table t4 mdam 'on';
--- SQL operation complete.
```

  To view compiler controls of type CT that are active, run the following query:

```
>>select * from table(compilercontrols()) where type = 'CT';
```

  The query displays the following output:

```
SEQNUM  TYPE   STATE   ATTRIBUTE  ATTRIBUTE_VALUE  TABLE_NAME

------ ----  ----    --------- ------------   ----------
0      CT   BY_USER  MDAM        ON             T3

0      CT   BY_USER  MDAM        ON             T4

--- 2 row(s) selected.
```

- Consider that the following CQS is set to ON:

```
>>CONTROL QUERY SHAPE JOIN (CUT,UNION(CUT,SCAN));
--- SQL operation complete.
```

  To view the control query shape, run the following query:

```
>>select * from table(compilercontrols()) where type =
'CQS'and state = 'BY_USER';
```

  The query displays the following output:

```
SEQNUM   TYPE  STATE  ATTRIBUTE   ATTRIBUTE_VALUE  TABLE_NAME

------   ---   ----   --------  -------------   ----------
0        CQS  BY_USER  CONTROL QUERY SHAPE  CONTROL QUERY
SHAPE JOIN(CUT,UNION(CUT,SCAN));  ?

--- 1 row(s) selected.
```

- Consider that the following CQS is set to ON:

```
>>CONTROL QUERY SHAPE SCAN (TABLE 'T1', PATH 'IT1');
--- SQL operation complete.
```

  To view the control query shape, run the following query:

```
>>select * from table(compilercontrols()) where type =
'CQS'and state = 'BY_USER';
```

The query displays the following output:

```
SEQNUM   TYPE  STATE  ATTRIBUTE   ATTRIBUTE_VALUE  TABLE_NAME

------   ---   ----   --------  -------------   ----------

0        CQS  BY_USER  CONTROL QUERY SHAPE  CONTROL QUERY
SHAPE SCAN (TABLE1 'T1', PATH 'IT1');  ?

 --- 1 row(s) selected.
```

CONCAT Function

The CONCAT function returns the concatenation of two character value expressions as a character string value. You can also use the concatenation operator (||).

CONCAT is an SQL/MX extension.

```
CONCAT (character-expr-1, character-expr-2)
```

character-expr-1, character-expr-2

are SQL character value expressions (of data type CHAR or VARCHAR) that specify two strings of characters. The result of the CONCAT function is the concatenation of *character-expr-1* with *character-expr-2*. See Character Value Expressions on page 6-41.

Concatenation Operator (||)

The concatenation operator, denoted by two vertical bars (||), concatenates two string values to form a new string value. To indicate that two strings are concatenated, connect the strings with two vertical bars (||):

character-expr-1 || *character-expr-2*

An operand can be any SQL value expression of data type CHAR or VARCHAR.

Considerations for CONCAT

Operands

A string value can be specified by any character value expression, such as a character string literal, character string function, column reference, aggregate function, scalar subquery, CASE expression, or CAST expression. The value of the operand must be of type CHAR or VARCHAR.

If you use the CAST expression, you must specify the length of CHAR or VARCHAR.

SQL Parameters

You can concatenate an SQL parameter and a character value expression. The concatenated parameter takes on the data type attributes of the character value expression. Consider this example, where `?p` is assigned a string value of `'5 March'`:

```
?p || ' 2002'
```

The type assignment of the parameter `?p` becomes CHAR(5), the same data type as the character literal `' 2002'`. Because you assigned a string value of more than five characters to `?p`, NonStop SQL/MX returns a truncation warning, and the result of the concatenation is `5 Mar 2002`.

To specify the type assignment of the parameter, use the CAST expression on the parameter as:

```
CAST(?p AS CHAR(7)) || '2002'
```

In this example, the parameter is not truncated, and the result of the concatenation is `5 March 2002`.

Examples of CONCAT

Suppose that the LOGFILE table has been created in NonStop SQL/MP as:

```
CREATE TABLE $sys.stats.logfile
 ( user_key  NUMERIC (3) UNSIGNED NO DEFAULT NOT NULL
  ,run_date  DATE DEFAULT CURRENT
  ,run_time  TIME DEFAULT CURRENT
  ,user_name VARCHAR (20)
  ,user_info VARCHAR (80)
  ,PRIMARY KEY   (user_key))
   CATALOG $sys.stats
   ORGANIZATION KEY SEQUENCED;
```

After the table is created, you can insert the mapping into the OBJECTS table in this way by using MXCI:

```
CREATE SQLMP ALIAS sys.stats.logfile $sys.stats.logfile;
```

- Insert information consisting of a single character string before you end an MXCI session. Use the CONCAT function to construct and insert the value:

```
INSERT INTO stats.logfile
(user_key, user_info)
VALUES (001, CONCAT ('Executed at ',
        CAST (CURRENT_TIMESTAMP AS CHAR(26))));
```

- Use the concatenation operator `||` to construct and insert the value:

```
INSERT INTO stats.logfile
(user_key, user_info)
VALUES (002, 'Executed at ' ||
        CAST (CURRENT_TIMESTAMP AS CHAR(26)));
```

This table now includes:

```
  1  2000-01-03  12:58:32            ?
Executed at 2000-01-03 12:58:32.527117

  2  2000-01-03  12:58:40            ?
Executed at 2000-01-03 12:58:40.364611
```

CONVERTTIMESTAMP Function

The CONVERTTIMESTAMP function converts a Julian timestamp to a value with data type TIMESTAMP.

CONVERTTIMESTAMP is an SQL/MX extension.

```
CONVERTTIMESTAMP (julian-timestamp)
```

julian-timestamp

is an expression that evaluates to a Julian timestamp, which is a LARGEINT value.

Considerations for CONVERTTIMESTAMP

Relationship to the JULIANTIMESTAMP Function

The operand of CONVERTTIMESTAMP is a Julian timestamp, and the function result is a value of data type TIMESTAMP. The operand of the JULIANSTAMP function is a value of data type TIMESTAMP, and the function result is a Julian timestamp. That is, the two functions have an inverse relationship to one another.

Use of CONVERTTIMESTAMP

You can use the inverse relationship between the JULIANTIMESTAMP and CONVERTTIMESTAMP functions to insert Julian timestamp columns into your database and display these column values in a TIMESTAMP format.

Examples of CONVERTTIMESTAMP

- Suppose that the EMPLOYEE table includes a column, named HIRE_DATE, which contains the hire date of each employee as a Julian timestamp. Convert the Julian timestamp into a TIMESTAMP value:

  ```
  SELECT CONVERTTIMESTAMP (hire_date)
  FROM persnl.employee;
  ```

- This example illustrates the inverse relationship between JULIANTIMESTAMP and CONVERTTIMESTAMP.

  ```
  SELECT CONVERTTIMESTAMP (JULIANTIMESTAMP (ship_timestamp))
  FROM persnl.project;
  ```

  If, for example, the value of SHIP_TIMESTAMP is 1998-04-03 21:05:36.143000, the result of `CONVERTTIMESTAMP(JULIANTIMESTAMP(ship_timestamp))` is the same value, 1998-04-03 21:05:36.143000.

COS Function

The COS function returns the cosine of a numeric value expression, where the expression is an angle expressed in radians.

COS is an SQL/MX extension.

```
COS (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the COS function. See Numeric Value Expressions on page 6-52.

Examples of COS

- This function returns the value 9.39680940386503936E-001, or approximately 0.9397, the cosine of 0.3491 (which is 20 degrees):

  ```
  COS (0.3491)
  ```

COSH Function

The COSH function returns the hyperbolic cosine of a numeric value expression, where the expression is an angle expressed in radians.

COSH is an SQL/MX extension.

```
COSH (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the COSH function. See Numeric Value Expressions on page 6-52.

Examples of COSH

- This function returns the value 1.88842387716101616E+000, or approximately 1.8884, the hyperbolic cosine of 1.25 in radians:

  ```
  COSH (1.25)
  ```

COUNT Function

The COUNT function counts the number of rows that result from a query or the number of rows that contain a distinct value in a specific column. The result of COUNT is data type LARGEINT. The result can never be NULL.

```
COUNT  {(*) | ([ALL | DISTINCT] expression)}
```

`COUNT (*)`

returns the number of rows in the table specified in the FROM clause of the SELECT statement that contains COUNT (*). If the result table is empty (that is, no rows are returned by the query) COUNT (*) returns zero.

`ALL | DISTINCT`

returns either the number of all rows or the number of distinct rows in the one-column table derived from the evaluation of *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the COUNT function is applied.

expression

specifies a value expression that determines the values to count. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the COUNT function operates on distinct values from the one-column table derived from the evaluation of *expression*. See Expressions on page 6-41.

Considerations for COUNT

Operands of the Expression

The operand of COUNT is either * or an expression that includes columns from the result table specified by the SELECT statement that contains COUNT. However, the expression cannot include an aggregate function or a subquery. These expressions are valid:

```
COUNT (*)
COUNT (DISTINCT JOBCODE)
COUNT (UNIT_PRICE * QTY_ORDERED)
```

Nulls

COUNT is evaluated after eliminating all nulls from the one-column table specified by the operand. If the table has no rows, COUNT returns zero.

COUNT(*) does not eliminate null rows from the table specified in the FROM clause of the SELECT statement. If all rows in a table are null, COUNT(*) returns the number of rows in the table.

Examples of COUNT

- Count the number of rows in the EMPLOYEE table:

```
SELECT COUNT (*)
FROM persnl.employee;

(EXPR)
-----------
         62

--- 1 row(s) selected.
```

- Count the number of employees who have a job code in the EMPLOYEE table:

```
SELECT COUNT (jobcode)
FROM persnl.employee;

(EXPR)
-----------
         56

--- 1 row(s) selected.

SELECT COUNT(*)
FROM persnl.employee
WHERE jobcode IS NOT NULL;

(EXPR)
-----------
         56

--- 1 row(s) selected.
```

- Count the number of distinct departments in the EMPLOYEE table:

```
SELECT COUNT (DISTINCT deptnum)
FROM persnl.employee;

(EXPR)
-----------
         11

--- 1 row(s) selected.
```

CURRENT Function

The CURRENT function returns a value of type TIMESTAMP based on the current local date and time. You can also use CURRENT_TIMESTAMP Function on page 8-43.

```
CURRENT [(precision)]
```

precision

is an integer value in the range 0 to 6 that specifies the precision of (the number of decimal places in) the fractional seconds in the returned value. The default is 6.

For example, the function CURRENT (2) returns the current date and time as a value of data type TIMESTAMP, where the precision of the fractional seconds is 2—for example, 1997-06-26 09:01:20.89. The value returned is not a string value.

Examples of CURRENT

- The PROJECT table contains a column SHIP_TIMESTAMP of data type TIMESTAMP. Update a row by using the CURRENT value:

```
UPDATE persnl.project
SET ship_timestamp = CURRENT
WHERE projcode = 1000;
```

CURRENT_DATE Function

The CURRENT_DATE function returns the local current date as a value of type DATE.

```
CURRENT_DATE
```

The CURRENT_DATE function returns the current date, such as 1997-09-28. The value returned is a value of type DATE, not a string value.

Examples of CURRENT_DATE

- Select rows from the ORDERS table based on the current date:

```
SELECT * FROM sales.orders
WHERE deliv_date >= CURRENT_DATE;
```

- The PROJECT table has a column EST_COMPLETE of type INTERVAL DAY. If the current date is the start date of your project, determine the estimated date of completion:

```
SELECT projdesc, CURRENT_DATE + est_complete
FROM persnl.project;

Project/Description  (EXPR)
-------------------  ----------
SALT LAKE CITY       2000-01-18
ROSS PRODUCTS        2000-02-02
MONTANA TOOLS        2000-03-03
AHAUS TOOL/SUPPLY    2000-03-03
THE WORKS            2000-02-02
THE WORKS            2000-02-02

--- 6 row(s) selected.
```

CURRENT_TIME Function

The CURRENT_TIME function returns the current local time as a value of type TIME.

```
CURRENT_TIME [(precision)]
```

precision

is an integer value in the range 0 to 6 that specifies the precision of (the number of decimal places in) the fractional seconds in the returned value. The default is 0.

For example, the function CURRENT_TIME (2) returns the current time as a value of data type TIME, where the precision of the fractional seconds is 2—for example, 14:01:59.30. The value returned is not a string value.

Examples of CURRENT_TIME

Suppose that the LOGFILE table has been created in NonStop SQL/MP as:

```
CREATE TABLE $sys.stats.logfile
 ( user_key  NUMERIC (3) UNSIGNED NO DEFAULT NOT NULL
  ,run_date  DATE
  ,run_time  TIME
  ,user_name VARCHAR (20)
  ,user_info VARCHAR (80)
  ,PRIMARY KEY   (user_key))
   CATALOG $sys.stats
   ORGANIZATION KEY SEQUENCED;
```

After the table is created, you can insert the mapping into the OBJECTS table in this way by using MXCI:

```
CREATE SQLMP ALIAS sys.stats.logfile $sys.stats.logfile;
```

- Use CURRENT_DATE and CURRENT_TIME as a value in an inserted row:

```
INSERT INTO stats.logfile
(user_key, run_date, run_time, user_name)
VALUES (001, CURRENT_DATE, CURRENT_TIME, 'JuBrock');
```

CURRENT_TIMESTAMP Function

The CURRENT_TIMESTAMP function returns a value of type TIMESTAMP based on the current local date and time. You can also use the CURRENT Function on page 8-40.

```
CURRENT_TIMESTAMP [(precision)]
```

precision

is an integer value in the range 0 to 6 that specifies the precision of (the number of decimal places in) the fractional seconds in the returned value. The default is 6.

For example, the function CURRENT_TIMESTAMP (2) returns the current date and time as a value of data type TIMESTAMP, where the precision of the fractional seconds is 2; for example, 1997-06-26 09:01:20.89. The value returned is not a string value.

Examples of CURRENT_TIMESTAMP

- The PROJECT table contains a column SHIP_TIMESTAMP of data type TIMESTAMP. Update a row by using the CURRENT_TIMESTAMP value:

```
UPDATE persnl.project
SET ship_timestamp = CURRENT_TIMESTAMP
WHERE projcode = 1000;
```

CURRENT_USER Function

The CURRENT_USER function returns the current Guardian user ID as variable-length character data in the form *group.name*.

```
CURRENT_USER
```

The CURRENT_USER function is equivalent to the SESSION_USER Function on page 8-168 and the USER Function on page 8-203.

Examples of CURRENT_USER

- Retrieve the user name value for the current user:

```
SELECT CURRENT_USER FROM logfile;

(EXPR)
-----------------------
DCS.TSHAW
...

--- 5 row(s) selected.
```

DATE_ADD Function

The DATE_ADD function adds the interval specified by the *interval_expression* to the *datetime_expression*.

```
DATE_ADD (datetime_expression, interval_expression)
```

datetime_expression

is an expression that specifies a datetime value of type, DATE, TIMESTAMP, or TIME.

interval_expression

is an expression that specifies an INTERVAL data type.

Return type

The return type is same as the type of *datetime_expression*, but if the *interval_expression* consists of any time components, a timestamp is returned. For example:

```
DATE_ADD(DATE ‘2007-02-28’, INTERVAL ‘7’ DAY) returns DATE
DATE_ADD(DATE ‘2007-02-28’, INTERVAL ‘7’ HOUR) returns TIMESTAMP
```

Examples of DATE_ADD

The following example illustrates the use of DATE_ADD function:

```
>>select dt_ts, date_add(dt_ts, interval '1' day) from T01
where sm_int = 2;

DT_TS                       (EXPR)

--------------------------  --------------------------


2010-02-28 20:13:53.000000  2010-03-01 20:13:53.000000


--- 1 row(s) selected.
```

DATEADD Function

The DATEADD function adds the *num_expression* units specified by the *datepart* to the *datetime_expression*.

```
DATEADD(datepart, num_expression, datetime_expression)
```

Note. Unlike DATE_ADD function, the interval is provided in terms of *datepart* and *num_expression* instead of an *interval_expression***.**

datepart

specifies the units of time being added: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, QUARTER, WEEK, or one of the following abbreviations:

YEAR	*YY* and *YYYY*
MONTH	*M* and *MM*
DAY	*D* and *DD*
HOUR	*HH*
MINUTE	*MI*
SECOND	*S* and *SS*
QUARTER	*Q* and *QQ*
WEEK	*WW* and *WK*

num_expression

is an SQL exact numeric value expression that specifies how many *datepart* units of time must be added to the *datetime_expression*.

datetime_expression

is an expression that specifies a datetime value of type, DATE, TIMESTAMP, or TIME.

Considerations for DATEADD

- If the specified interval is in years or months, DATEADD normalizes the result. For example, if the end day of the resulting date is invalid, the day will be rounded off to the last day of the resulting month.
- If *num_expression* has a fraction, it is ignored. However, if *datepart* is SECOND, then the fractional portion for *num_expression* is considered.
- If *num_expression* is negative, *num_expression* units of *datepart* are subtracted from *datetime_expression*.

Return type

The return type is same as the *datetime_expression* type, but if the *datetime_expression* consists of any time components, a timestamp is returned. For example:

```
DATEADD(DAY, 7, ‘DATE 2007-02-28’) returns DATE
DATEADD(HOUR, 7, ‘DATE 2007-02-28’) returns TIMESTAMP
```

Examples of DATEADD

The following example illustrates the use of DATEADD function:

```
>>select dt_ts, dateadd(week, 5, dt_ts) from T01;

DT_TS                         (EXPR)

--------------------------    --------------------------


2008-11-12 03:03:03.000000    2008-12-17 03:03:03.000000

2112-01-30 09:05:00.000000    2112-03-05 09:05:00.000000


--- 2 row(s) selected.
```

DATEDIFF Function

The DATEDIFF function returns the integer number of *datepart* unit boundaries crossed between the *startdate* and *enddate*.

DATEDIFF (*datepart*, *startdate*, *enddate*)

datepart

represents YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, QUARTER, WEEK, or one of the following abbreviations:

YEAR	*YY* and *YYYY*
MONTH	*M* and *MM*
DAY	*D* and *DD*
HOUR	*HH*
MINUTE	*MI*

SECOND	*S* and *SS*
QUARTER	*Q* and *QQ*
WEEK	*WW* and *WK*

startdate, enddate

is of type DATE or TIMESTAMP.

Considerations for DATEDIFF

- If *startdate* is later than *enddate*, the return value is either negative or zero.
- If the result is out of range for integer values, an error is returned.
- The maximum seconds *datepart* allowed is equivalent to 68 years.
- If a difference in weeks is specified, and one of the two dates is earlier than January 7, 0001, then an error is returned.

Return type

The result is a signed integer value equal to the number of *datepart* boundaries crossed between the first and second date.

Examples of DATEDIFF

The following example illustrates the use of DATEDIFF function:

```
>>select dt_ts, dt_date, datediff(second, dt_ts, dt_date)
from T01;

DT_TS                       DT_DATE     (EXPR)
--------------------------  ----------  -----------

2008-11-12 03:03:03.000000  2011-11-27     95893017
2112-01-30 09:05:00.000000  2112-01-30       -32700

--- 2 row(s) selected.
```

DATEFORMAT Function

The DATEFORMAT function returns a datetime value as a character string literal in the DEFAULT, USA, or EUROPEAN format. The data type of the result is CHAR.

DATEFORMAT is an SQL/MX extension.

```
DATEFORMAT (datetime-expression,{DEFAULT | USA | EUROPEAN})
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE, TIME, or TIMESTAMP. See Datetime Value Expressions on page 6-43.

`DEFAULT | USA | EUROPEAN`

specifies a format for a datetime value. See Datetime Literals on page 6-68.

Examples of DATEFORMAT

- Convert a datetime literal in DEFAULT format to a string in USA format:

  ```
  DATEFORMAT (TIMESTAMP '1996-06-20 14:20:20.00', USA)
  ```

  The function returns this string literal:

  ```
  '06/20/1996 02:20:20.00 PM'
  ```

- Convert a datetime literal in DEFAULT format to a string in European format:

  ```
  DATEFORMAT (TIMESTAMP '1996-06-20 14:20:20.00', EUROPEAN)
  ```

  The function returns this string literal:

  ```
  '20.06.1996 14.20.20.00'
  ```

DATE_SUB Function

The DATE_SUB function subtracts the specified *interval-expression* from the *datetime-expression*.

```
DATE_SUB (datetime_expression, interval_expression)
```

datetime-expression

is an expression that specifies a datetime value of type DATE, TIMESTAMP, or TIME.

interval-expression

is an expression that specifies an INTERVAL data type.

Considerations for DATE_SUB

If the specified interval is in years or months, the DATE_SUB function normalizes the result. For example, if the end day of the resulting date is invalid, the day is rounded off to the last day of the resulting month.

Return type

The return type is same as the type of *datetime-expression*, but if the *interval-expression* consists of any time components, a timestamp is returned. For example,

```
DATE_SUB(DATE '2007-02-28', INTERVAL '7' DAY) returns DATE
DATE_SUB(DATE '2007-02-28', INTERVAL '7' HOUR) returns TIMESTAMP
```

Examples of DATE_SUB

The following example illustrates the use of DATE_SUB function:

```
>>select dt_ts, intr_sec, date_sub(dt_date, intr_sec) from
T01;


DT_TS                       INTR_SEC    (EXPR)

--------------------------  ----------  ---------------------
-----

2008-11-12 03:03:03.000000    7.000000  2011-11-26
23:59:53.0000002112-01-29 23:58:50.000000

2112-01-30 09:05:00.000000   70.000000  2112-01-29
23:58:50.000000


--- 2 row(s) selected.
```

DAY Function

The DAY function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 31 that represents the corresponding day of the month. The result returned by the DAY function is equal to the result returned by the DAYOFMONTH function.

DAY is an SQL/MX extension.

```
DAY (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of DAY

- Return an integer that represents the day of the month from the START_DATE column of the PROJECT table:

```
SELECT start_date, ship_timestamp, DAY(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
1996-04-10  1996-04-21 08:15:00.000000      10
```

DAYNAME Function

The DAYNAME function converts a DATE or TIMESTAMP expression into a character literal that is the name of the day of the week (Sunday, Monday, and so on).

DAYNAME is an SQL/MX extension.

```
DAYNAME (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of DAYNAME

- Return the name of the day of the week from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, DAYNAME(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ---------
1996-04-10  1996-04-21 08:15:00.000000  Wednesday
```

DAYOFMONTH Function

The DAYOFMONTH function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 31 that represents the corresponding day of the month. The result returned by the DAYOFMONTH function is equal to the result returned by the DAY function.

DAYOFMONTH is an SQL/MX extension.

```
DAYOFMONTH (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of DAYOFMONTH

- Return an integer that represents the day of the month from the START_DATE column of the PROJECT table:

```
SELECT start_date, ship_timestamp, DAYOFMONTH(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
1996-04-10  1996-04-21 08:15:00.000000      10
```

DAYOFWEEK Function

The DAYOFWEEK function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 7 that represents the corresponding day of the week. The value 1 represents Sunday, 2 represents Monday, and so forth.

DAYOFWEEK is an SQL/MX extension.

```
DAYOFWEEK (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of DAYOFWEEK

- Return an integer that represents the day of the week from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, DAYOFWEEK(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
1996-04-10  1996-04-21 08:15:00.000000       4
```

The value returned is 4, representing Wednesday. The week begins on Sunday.

DAYOFYEAR Function

The DAYOFYEAR function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 366 that represents the corresponding day of the year.

DAYOFYEAR is an SQL/MX extension.

```
DAYOFYEAR (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of DAYOFYEAR

- Return an integer that represents the day of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, DAYOFYEAR(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped               (EXPR)
----------  -------------------------  --------------------
1996-04-10  1996-04-21 08:15:00.000000                  101
```

DECODE Function

The functionality of DECODE is similar to the CASE-WHEN-THEN-ELSE-END expression. The DECODE function compares the `expr` with each `test_expr` one by one in the order specified. If `expr` is equal to a `test_expr,` the function returns the corresponding `retval` value. If no match is found, `default` is returned. If no match is found and `default` is omitted, NULL is returned.

```
DECODE (expr,test_expr,retval [,test_expr2,retval2 ... ] [ ,
default] )
```

`expr,test_expr,test_expr2` are SQL value expressions of comparable data types.

`retval` is a SQL value expression.

`default, retval, retval2,..` are SQL value expressions of comparable data types.

`DECODE (`*`expr,test_expr,retval [,test_expr2,retval2 ... ] [ , default]`*`)` is logically equivalent to the following:

```
CASE WHEN (expr IS NULL AND test_expr IS NULL) OR
expr = test_expr THEN retval
WHEN (expr IS NULL AND test_expr2 IS NULL) OR
expr = test_expr2 THEN retval2
...
ELSE default /* or ELSE NULL if default not specified */
END
```

Considerations

- In a DECODE function, two NULLs are considered to be equivalent. If `expr` is NULL, then the returned value is the `retval` of the first `test_expr` that is also NULL.
- The arguments can be any of the numeric types or character types. However, `expr` and each `test_expr` value must be of comparable data types. If `expr` and `test_expr` values are character types, they must be in the same character set (to be comparable types).
- All the `default` and `retval` value, if any, must be of comparable types.
- If `expr` and a `test_expr` value are character data, the comparison is made using nonpadded comparison semantics.
- If `expr` and a `test_expr` value are numeric data, the comparison is made with a temporary copy of one of the numbers, according to the NonStop SQL/MX defined rules of conversion. For example, if one number is integer and the other is decimal, the comparison is made with a temporary copy of the integer converted to a decimal.

- If all the possible return values are fixed-length character types, the return value is a fixed-length character string equal to the maximum size of all the possible return value types.
- If any of the possible return values are variable-length character types, the return value is a variable-length character string with maximum size of all the possible return value types.
- If all the possible return values are integer types, the return value is of the same data type as the largest integer type of all the possible return value types.
- If the return value is FLOAT, the precision is the maximum precision of all the possible return values.
- If all the possible return values are of the same non-integer, numeric type (REAL, FLOAT, DOUBLE PRECISION, NUMERIC, or DECIMAL), the return value is of the same type.
- If all the possible return values are numeric types but not all the same, and at least one is REAL, FLOAT, or DOUBLE PRECISION, the return value is a DOUBLE PRECISION type.
- If all the possible return values are numeric types but not all the same, none are REAL, FLOAT, or DOUBLE PRECISION, and at least one is of type NUMERIC, the return value is NUMERIC type.
- If all the possible return values are numeric types, none are NUMERIC, REAL, FLOAT, or DOUBLE PRECISION, and at least one is of type DECIMAL, the return value is DECIMAL type.
- If the return value is of type NUMERIC or DECIMAL, the value has a precision value equal to the sum of:
    - The maximum scale of all the return values.
    - The maximum of precision minus scale value for all the return values.

      However, the precision value must not exceed 18.
- If the return value is of type NUMERIC or DECIMAL, the scale of the return value is the minimum of:
    - The maximum scale of all the return values.
    - 18 - (the maximum of precision minus scale value for all the return values).
- There is no limit on the number of components in the DECODE function (includes `expr,test_exprs,retval`, and the `default`) other than the general limit of an SQL expression. However, large lists of components do not perform well.

Note. Existing NonStop SQL/MX conversions rules are applied to the above rules**.**

Examples of DECODE

- In this example, the DECODE function returns a value of 3:

```
create table tab1( colA varchar(15));

insert into tab1 values('FAILED');

select DECODE(colA ,'PREPARE',1,

'OPERATION',2,

'FAILED',3,

'SUCCESS',4,

0)

from tab1;

(EXPR)
----------

3
```

DEGREES Function

The DEGREES function converts a numeric value expression expressed in radians to the number of degrees.

DEGREES is an SQL/MX extension.

```
DEGREES (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the DEGREES function. See Numeric Value Expressions on page 6-52.

Examples of DEGREES

- This function returns the value 45 in degrees:

```
DEGREES (0.78540)
```

- This function returns the value 45. The function DEGREES is the inverse of the function RADIANS.

```
DEGREES (RADIANS (45))
```

DIFF1 Function

Considerations for DIFF1
Examples of DIFF1

The DIFF1 function is a sequence function that calculates the amount of change in an expression from row to row in an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

DIFF1 is an SQL/MX extension.

```
DIFF1 (column-expression-a [,column-expression-b])
```

column-expression-a

specifies a derived column determined by the evaluation of the column expression. If you specify only one column as an argument, DIFF1 returns the difference between the value of the column in the current row and its value in the previous row; this version calculates the unit change in the value from row to row.

column-expression-b

specifies a derived column determined by the evaluation of the column expression. If you specify two columns as arguments, DIFF1 returns the difference in consecutive values in *column-expression-a* divided by the difference in consecutive values in *column-expression-b*.

The purpose of the second argument is to distribute the amount of change from row to row evenly over some unit of change (usually time) in another column.

Considerations for DIFF1

Equivalent Result

If you specify one argument, the result of DIFF1 is equivalent to:

```
column-expression-a - OFFSET(column-expression-a, 1)
```

If you specify two arguments, the result of DIFF1 is equivalent to:

```
DIFF1(column-expression-a) / DIFF1(column-expression-b)
```

The two-argument version involves division by the result of the DIFF1 function. To avoid divide-by-zero errors, make sure that *column-expression-b* does not contain any duplicate values whose DIFF1 computation could result in a divisor of zero.

Datetime Arguments

In general, NonStop SQL/MX does not allow division by a value of INTERVAL data type. However, to permit use of the two-argument version of DIFF1 with times and dates, NonStop SQL/MX relaxes this restriction and allows division by a value of INTERVAL data type.

Examples of DIFF1

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	**TS**
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
19058	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Retrieve the difference between the I1 column in the current row and the I1 column in the previous row:

  ```
  SELECT DIFF1 (I1) AS DIFF1_I1
  FROM mining.seqfcn
  SEQUENCE BY TS;

  DIFF1_I1
  ------------
             ?
         21959
         -9116
        -14461
          7369

  --- 5 row(s) selected.
  ```

  Note that the first row retrieved displays null because the offset from the current row does not fall within the results set.

- Retrieve the difference between the TS column in the current row and the TS column in the previous row:

```
SELECT DIFF1 (TS) AS DIFF1_TS
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF1_TS
--------------------
                   ?
     30002620.000000
    134157861.000000
    168588029.000000
    114055223.000000

--- 5 row(s) selected.
```

Note that the results are expressed as the number of seconds. For example, the difference between TIMESTAMP '1951-02-15 14:35:49' and TIMESTAMP '1950-03-05 08:32:09' is approximately 347 days. The difference between TIMESTAMP '1955-05-18 08:40:10' and TIMESTAMP '1951-02-15 14:35:49' is approximately 4 years and 3 months, and so on.

- This query retrieves the difference in consecutive values in I1 divided by the difference in consecutive values in TS:

```
SELECT DIFF1 (I1,TS) AS DIFF1_I1TS
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF1_I1TS
-------------------
                  ?
           .0007319
          -.0000679
          -.0000857
           .0000646

--- 5 row(s) selected.
```

Note that the results are equivalent to the quotient of the results from the two preceding examples. For example, in the second row of the output of this example, 0.0007319 is equal to 21959 divided by 30002620.

DIFF2 Function

Considerations for DIFF2
Examples of DIFF2

The DIFF2 function is a sequence function that calculates the amount of change in a DIFF1 value from row to row in an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

DIFF2 is an SQL/MX extension.

```
DIFF2 (column-expression-a [,column-expression-b])
```

column-expression-a

specifies a derived column determined by the evaluation of the column expression. If you specify only one column as an argument, DIFF2 returns the difference between the value of DIFF1(*column-expression-a*) in the current row and the same result in the previous row.

column-expression-b

specifies a derived column determined by the evaluation of the column expression. If you specify two columns as arguments, DIFF2 returns the difference in consecutive values of DIFF1(*column-expression-a*) divided by the difference in consecutive values in *column-expression-b*.

See DIFF1 Function on page 8-59.

Considerations for DIFF2

Equivalent Result

If you specify one argument, the result of DIFF2 is equivalent to:

```
DIFF1(column-expression-a)- OFFSET(DIFF1(column-expression-a),1)
```

If you specify two arguments, the result of DIFF2 is equivalent to:

```
DIFF2(column-expression-a) / DIFF1(column-expression-b)
```

The two-argument version involves division by the result of the DIFF1 function. To avoid divide-by-zero errors, make sure that *column-expression-b* does not contain any duplicate values whose DIFF1 computation could result in a divisor of zero.

Datetime Arguments

In general, NonStop SQL/MX does not allow division by a value of INTERVAL data type. However, to permit use of the two-argument version of DIFF2 with times and dates, NonStop SQL/MX relaxes this restriction and allows division by a value of INTERVAL data type.

Examples of DIFF2

Suppose that SEQFCN has been created as:

```
CREATE TABLE mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by the TS column:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
19058	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Retrieve the difference between the value of DIFF1(I1) in the current row and the same result in the previous row:

```
SELECT DIFF2 (I1) AS DIFF2_I1
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF2_I1
--------------------
                   ?
                   ?
              -31075
               -5345
               21830

--- 5 row(s) selected.
```

  Note that the results are equal to the difference of DIFF1(I1) for the current row and DIFF1(I1) of the previous row. For example, in the third row of the output of this example, -31075 is equal to -9116 minus 21959. The value -9116 is the result of DIFF1(I1) for the current row, and the value 21959 is the result of DIFF1(I1) for the previous row. See Examples of DIFF1 on page 8-60.

- Retrieve the difference in consecutive values of DIFF1(I1) divided by the difference in consecutive values of TS:

```
SELECT DIFF2 (I1,TS) AS DIFF2_I1TS
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF2_I1TS
---------------------
                    ?
                    ?
             -.000231
             -.000031
              .000191

--- 5 row(s) selected.
```

EXP Function

This function returns the exponential value (to the base e) of a numeric value expression.

EXP is an SQL/MX extension.

```
EXP (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the EXP function. See Numeric Value Expressions on page 6-52.

The minimum input value must be between -744.4400719 and -744.4400720.

The maximum input value must be between 709.78271289338404 and 709.78271289338405.

Examples of EXP

- This function returns the value 3.49034295746184208E+000, or approximately 3.4903:

  ```
  EXP (1.25)
  ```

- This function returns the value 2.0. The function EXP is the inverse of the function LOG:

  ```
  EXP (LOG(2.0))
  ```

EXPLAIN Function

Considerations for EXPLAIN
Examples of EXPLAIN

The EXPLAIN function is a table-valued stored function that generates a result table describing an access plan for a SELECT, INSERT, DELETE, UPDATE, or CALL statement. See Result of the EXPLAIN Function on page 8-67.

The EXPLAIN function can be specified as a table reference (*table*) in the FROM clause of a SELECT statement if it is preceded by the keyword TABLE and surrounded by parentheses.

The EXPLAIN function is an SQL/MX extension.

```
EXPLAIN (module,'statement-pattern')

module is:
   'module-name'
MXCI | NULL ■
```

'*module-name*'

is a character string that specifies the full name of a prepared embedded SQL module, including the catalog name, schema name, and any module management attributes. See the MODULE Directive on page 3-70. For more information on module management attributes, see the *SQL/MX Programming Manual for C and COBOL*.

The module name is enclosed in single quotes and is case-sensitive. If a module name is uppercase, the value you specify within single quotes must be uppercase. For example, `'MYCAT.MYSCH.MYPROG'`

MXCI NULL

explains statements prepared in the MXCI session. ■

'*statement-pattern*'

is a character string that specifies the pattern for searching for the names of SQL statements within the given module. If the module is specified as NULL, the pattern string is used to match statement names that have been used in PREPARE statements within the current MXCI session.

A statement pattern is enclosed in single quotes and is case-sensitive. The statement name must be in uppercase, unless you delimit the statement name in a PREPARE statement. The pattern can include wild-card characters as in a LIKE pattern. See LIKE Predicate on page 6-97.

Considerations for EXPLAIN

Using a Statement Pattern

Using a statement pattern is analogous to using a LIKE pattern. For example, this statement returns the EXPLAIN result for all statements prepared within the current MXCI session:

```
SELECT * FROM TABLE (EXPLAIN (NULL,'%'))
```

This statement returns the EXPLAIN result for all statements prepared within the embedded SQL module named `MYCAT.MYSCH.MYPROG`:

```
SELECT * FROM TABLE (EXPLAIN ('MYCAT.MYSCH.MYPROG','%'))
```

This statement returns the EXPLAIN result for all prepared statements whose names begin with the uppercase letter `'S'`:

```
SELECT * FROM TABLE (EXPLAIN (NULL,'S%'))
```

If the statement pattern does not find any matching statement names, no rows are returned as the result of the SELECT statement.

For more information about module names, see the *SQL/MX Programming Manual for C and COBOL*.

Using EXPLAIN and EXPLAIN Statement

The result of the EXPLAIN function for a specific DML statement can be generated either by using the EXPLAIN function or the EXPLAIN statement. Use the EXPLAIN function only for prepared statements.

The EXPLAIN statement displays the result table of the EXPLAIN function with various formatting options. If you want to display only some of the columns, you must use the EXPLAIN function to return the intermediate result table that you then query with a SELECT statement.

Result of the EXPLAIN Function

The result table of the EXPLAIN function describes the access plans for SELECT, INSERT, DELETE, UPDATE, or CALL statements. Use the EXPLAIN function to generate the result and the EXPLAIN statement to display the result.

See the EXPLAIN Function on page 8-66 and EXPLAIN Statement on page 2-208.

In this description of the result of the EXPLAIN function, an operator tree is a structure that represents operators used in an access plan as nodes, with at most one parent node for each node in the tree, and with only one root node.

A node of an operator tree is a point in the tree that represents an event (involving an operator) in a plan. Each node might have subordinate nodes—that is, each event might generate a subordinate event or events in the plan.

Column Name	Data Type	Description
MODULE_NAME	CHAR(60)	Module name as specified in the argument to the EXPLAIN function; if NULL, it takes the name of the current module. MODULE_NAME shows DYNAMICALLY COMPILED when a query statement or prepared statement is supplied as the argument to the EXPLAIN statement.
STATEMENT_ NAME	CHAR(60)	Statement name after wild-card character expansion; truncated on the right if longer than 60 characters.
PLAN_ID	LARGEINT	Unique system-generated plan ID automatically assigned by NonStop SQL/MX; generated at compile time.
SEQ_NUM	INT	Sequence number of the current node in the operator tree; indicates the sequence in which the operator tree is generated.

Column Name	Data Type	Description
OPERATOR	CHAR(30)	Current node type; one of these:

Column Name	Data Type	Description	
		Operator	**Group (if any)**
		CALL	UDR
		CURSOR_DELETE	DAM unique
		CURSOR_UPDATE	DAM unique
		ESP_ACCESS	Sequence Generator
		ESP_EXCHANGE	Exchange
		EXPLAIN	Stored function
		EXPR	Tuple
		FILE_SCAN	DAM subset
		FILE_SCAN_UNIQUE	DAM unique
		HASH_GROUPBY	Groupby
		HASH_PARTIAL_GR..._LEAF	Groupby
		HASH_PARTIAL_GR..._ROOT	Groupby
		HYBRID_HASH_JOIN	Join
		HYBRID_HASH_SEMI_JOIN	Join
		HYBRID_HASH_ANTI_SEMI_JOIN	Join
		INDEX_SCAN	DAM subset
		INDEX_SCAN_UNIQUE	DAM unique
		INSERT	Insert
		INSERT_VSBB	Insert
		LEFT_HYBRID_HASH_JOIN	Join
		LEFT_MERGE_JOIN	Join
		LEFT_NESTED_JOIN	Join
		LEFT_ORDERED_HASH_JOIN	Join
		MATERIALIZE	Materialize
		MERGE_ANTI_SEMI_JOIN	Join
		MERGE_JOIN	Join
		MERGE_SEMI_JOIN	Join
		MERGE_UNION	Merge union
		NESTED_ANTI_SEMI_JOIN	Join
		NESTED_JOIN	Join
		NESTED_SEMI_JOIN	Join
		NextValueFor	Sequence Generator
		ORDERED_HASH_ANTI_SEMI_JOIN	Join
		ORDERED_HASH_JOIN	Join
		ORDERED_HASH_SEMI_JOIN	Join
		PACK	Rowset
		PARTITION_ACCESS	Exchange
		ROOT	Root
		SAMPLE	Data mining
		SEQUENCE	Data mining
		Sequence Generator	Sequence Generator
		SHORTCUT_SCALAR_AGGR	Groupby
		SORT	Sort
		SORT_GROUPBY	Groupby

Column Name	Data Type	Description	
		SORT_PARTIAL_AG..._LEAF	Groupby
		SORT_PARTIAL_AG..._ROOT	Groupby
		SORT_PARTIAL_GR..._LEAF	Groupby
		SORT_PARTIAL_GR..._ROOT	Groupby
		SORT_SCALAR_AGGR	Groupby
		SPLIT_TOP	Exchange
		SUBSET_DELETE	DAM subset
		SUBSET_UPDATE	DAM subset
		TRANSPOSE	Data mining
		TUPLE_FLOW	Join
		TUPLELIST	Tuple
		UNIQUE_DELETE	DAM unique
		UNIQUE_UPDATE	DAM unique
		UNPACK	Rowset
		VALUES	Tuple
LEFT_CHILD_ SEQ_NUM	INT	Sequence number for the first child operator of the current node (or operator); null if node has no child operators.	
RIGHT_CHILD_ SEQ_NUM	INT	Sequence number for the second child operator of the current node (or operator); null if node does not have a second child.	
TNAME	CHAR(60)	For operators in scan group, full name of base table, truncated on the right if too long for column. If correlation name differs from table name, simple correlation name first and then table name in parentheses.	
CARDINALITY	REAL	Estimated number of rows that will be returned by the current node.	
OPERATOR_ COST	REAL	Estimated cost associated with the current node to execute the operator.	
TOTAL_COST	REAL	Estimated cost associated with the current node to execute the operator, including the cost of all subtrees in the operator tree.	
DETAIL_COST	VARCHAR (200)	Tokenized cost vector.	
DESCRIPTION	VARCHAR (3000)	Additional information about the operation in the form of a stream of token pairs.	

Operators are grouped together for purposes of display within the EXPLAIN statement. For more information about the use of the result table of the EXPLAIN function, see the *SQL/MX Query Guide*.

Examples of EXPLAIN

- Use the EXPLAIN statement to construct and display all columns in the result table of the EXPLAIN function for the specified prepared statement:

```
prepare xx from
select * from part where p_partkey = (select max(ps_partkey)
from partsupp);


explain options 'f' xx;
```

  The EXPLAIN statement display for the prepared statement named xx is identical to the EXPLAIN statement display shown under FC Command on page 4-30.

- Display the specified columns in the result table of the EXPLAIN function for the same prepared statement FINDEMP:

```
SELECT SEQ_NUM, OPERATOR, OPERATOR_COST
FROM TABLE (EXPLAIN (NULL, 'FINDEMP'));

SEQ_NUM      OPERATOR                     OPERATOR_COST
-----------  ----------------------  --------------------
          1  FILE_SCAN                      1.6196700E-001
          2  PARTITION_ACCESS               4.3732533E-003
          3  ROOT                           1.0392011E-006

--- 3 row(s) selected.
```

  The preceding example displays only part of the result table of the EXPLAIN function. It first uses the EXPLAIN function to generate the table and then selects the desired columns.

- Display the specified columns in the result table of the EXPLAIN function for the same prepared statement but with two different plans. The first plan is the default plan generated by the optimizer, and the second plan is forced by using the CONTROL QUERY SHAPE statement.

  This SET SHOWSHAPE command displays the plan generated by the optimizer:

```
SET SHOWSHAPE ON;

PREPARE FINDEMP1 FROM
  SELECT last_name, first_name, deptnum,
    employee.jobcode, jobdesc
  FROM employee, job
  WHERE deptnum = 3100 AND employee.jobcode = job.jobcode;

control query shape merge_join(sort(
partition_access(scan('EMPLOYEE',forward,
blocks_per_access 1 , mdam off))),
partition_access(scan('JOB', forward,
blocks_per_access 3 , mdam off)));

SELECT SEQ_NUM, OPERATOR, OPERATOR_COST, TOTAL_COST
FROM TABLE (EXPLAIN (NULL, 'FINDEMP1'));
```

```
SEQ_NUM  OPERATOR          OPERATOR_COST    TOTAL_COST
-------  ----------------  ---------------  ---------------
      1  FILE_SCAN          1.6196700E-001   1.6196700E-001
      2  PARTITION_ACCESS   4.4135637E-003   1.6196700E-001
      3  SORT               1.9920971E-001   2.0409727E-001
      4  FILE_SCAN          1.6560700E-001   1.6560700E-001
      5  PARTITION_ACCESS   7.1685006E-003   1.6560700E-001
      6  MERGE_JOIN         2.0821783E-003   2.1525979E-001
      7  ROOT               2.7007004E-005   2.1528682E-001

--- 7 row(s) selected.
```

The second plan is forced by this CONTROL QUERY SHAPE statement:

```
control query shape nested_join(
partition_access(scan),
partition_access(scan('JOB')));

PREPARE FINDEMP2 FROM
  SELECT last_name, first_name, deptnum,
    employee.jobcode, jobdesc
  FROM employee, job
  WHERE deptnum = 3100 AND employee.jobcode = job.jobcode;

SELECT SEQ_NUM, OPERATOR, OPERATOR_COST, TOTAL_COST
FROM TABLE (EXPLAIN (NULL, 'FINDEMP2'));

SEQ_NUM  OPERATOR          OPERATOR_COST    TOTAL_COST
-------  ----------------  ---------------  ---------------
      1  FILE_SCAN          1.6196700E-001   1.6196700E-001
      2  PARTITION_ACCESS   4.4135637E-003   1.6196700E-001
      4  FILE_SCAN_UNIQUE   2.0590099E-001   2.0590099E-001
      5  PARTITION_ACCESS   4.5211268E-003   2.0590099E-001
      6  NESTED_JOIN        1.7425649E-005   3.6786800E-001
      7  ROOT               2.7007004E-005   3.6786800E-001

--- 6 row(s) selected.
```

You can compare the two result tables of the EXPLAIN function to determine which plan to use for this query. The total cost of the ROOT node indicates the total cost of the plan. Therefore, if you compare the two costs, the plan generated by the optimizer is the better plan, as reported by the EXPLAIN function.

- Display all columns in the result table of the EXPLAIN function for the CALL operator:

```
>>prepare S from call samdbcat.sales.order_summary(?, ?);

--- SQL command prepared.
>>select description from table(explain(NULL, 'S')) where
operator = 'CALL';

DESCRIPTION

-------------------------------------------------------------
```

```
parameter_modes: I O
routine_name: SAMDBCAT.SALES.ORDER_SUMMARY
routine_label: \ALPINE.$SYSTEM.ZSDCR2C6.L1Z7NW00
sql_access_mode: READS SQL DATA external_name: orderSummary2
external_path: /usr/mydir/myclasses external_file: rs
signature:
(Ljava/lang/String;[J[Ljava/sql/ResultSet;[Ljava/sql/ResultSe
t;)V
language: Java runtime_options: OFF
runtime_option_delimiters: ' '
max_results: 2

--- 1 row(s) selected.
```

EXTRACT Function

The EXTRACT function extracts a datetime field from a datetime or interval value expression. It returns an exact numeric value.

```
EXTRACT (datetime-field FROM extract-source)

datetime-field is:
   YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

extract-source is:
   datetime-expression | interval-expression
```

See Datetime Value Expressions on page 6-43 and Interval Value Expressions on page 6-47.

Examples of EXTRACT

- Extract the year from a DATE value:

  ```
  EXTRACT (YEAR FROM DATE '1996-09-28')
  ```

  The result is 1996.

- Extract the year from an INTERVAL value:

  ```
  EXTRACT (YEAR FROM INTERVAL '01-09' YEAR TO MONTH)
  ```

  The result is 1.

FEATURE_VERSION_INFO Function

FEATURE_VERSION_INFO is a built-in table-valued function that returns feature version information for all user objects with an object feature version (OFV) higher than a given value, in a specified set of catalogs. Information is not returned for definition schema tables or user metadata tables.

```
feature_version_info ('E_TYPE', 'E_VALUE', 'E_VERSION')
```

Input and Output Parameters

Table 8-1 shows the input and output parameters for FEATURE_VERSION_INFO.

Table 8-1. Input and Output Parameters for FEATURE_VERSION_INFO

Input/Output Type	Parameter	Specification	Description
Input parameter	E_TYPE	CHAR (32) NOT NULL	The type of version information that is desired.
Input parameter	E_VALUE	VARCHAR(518) NOT NULL	The name of the entity for which version information is desired. The type of that entity is implied by E_TYPE.
Input parameter	E_VERSION	INT NOT NULL	The target feature version.
Output column	E_TYPE	CHAR (32) NOT NULL	A copy of the actual value for the E_TYPE input parameter.
Output column	E_VALUE	VARCHAR(518) NOT NULL	A copy of the actual value for the E_VALUE input parameter.
Output column	E_VERSION	INT NOT NULL	A copy of the actual value for the E_VALUE input parameter.
Output column	OBJECT_NAME	VARCHAR(776) NOT NULL	The fully qualified external format ANSI name of a database object with OFV higher than E_VERSION.
Output column	OBJECT_TYPE	CHAR(2) NOT NULL	The two character object type for the affected database object.
Output column	FEATURE_VERSION	INT NOT NULL	The actual OFV of that database object.

Note. Possible values for the E_TYPE input parameter are:

- CATALOG – The E_VALUE parameter specifies the external format ANSI name of a catalog. Output rows are for objects in that catalog only.
- CATALOG_CASCADE – The E_VALUE parameter specifies the external format ANSI name of a catalog. Output rows are for objects in that catalog and catalogs that are related to it.

Example of FEATURE_VERSION_INFO

```
select object_name, feature_version
  from table (feature_version_info ('CATALOG', 'CATX', 1200));

OBJECT_NAME                                 FEATURE_VERSION
----------------------------------------    ---------------
CATX."schema x"."table with large key"      3000
CATX.SCHEMAY."table with bignum column"     3000
         ...
```

FLOOR Function

The FLOOR function returns the largest integer, represented as a FLOAT data type, less than or equal to a numeric value expression.

FLOOR is an SQL/MX extension.

```
FLOOR (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the FLOOR function. See Numeric Value Expressions on page 6-52.

Examples of FLOOR

- This function returns the integer value 2.00000000000000040E+000, represented as a FLOAT data type:

  ```
  FLOOR (2.25)
  ```

HASHPARTFUNC Function

HashPartFunc is the function NonStop SQL/MX uses to hash partition data. HashPartFunc returns the number of the partition to which a row identified by the specified partitioning key would belong if the table were hash partitioned.

HashPartFunc is an SQL/MX extension.

```
HashPartFunc(partitioning-key FOR num-partitions)
```

partitioning-key

is the prospective partitioning key value of a row from a partitioned or nonpartitioned table. *partitioning-key* is a comma-separated list of values that make up the partitioning key.

num-partitions

is the number of partitions that you might create in the future, not the number in the table from which the rows are currently being read.

Considerations for HashPartFunc

Cast the partitioning key values to their declared types, because HashPartFunc is sensitive to the data type.

HashPartFunc is evaluated based only on the variables you enter rather than the underlying table. You can use this function on a nonpartitioned table to find out what the data distribution would be if you were to hash partition the table in various ways; that is, with different numbers of partitions and different partitioning keys.

You can also use HashPartFunc to preorder data for efficient insertion into a hash-partitioned table with a specified number of partitions. The most efficient insertion times usually are achieved when both of these conditions are met:

- All of the data rows destined for a particular partition are grouped together.
- Within each grouping, data rows are sorted by the clustering key of the destination table. (The clustering key is typically the same as the primary key, but it might be different if, for example, the destination table is created with a STORE BY key that is different from the primary key.)

When you use HashPartFunc, partition numbers are mapped to the physical partitions in the order in which the partitions will be added when the desired destination table is created with the CREATE TABLE statement.

Examples of HashPartFunc

- HashPartFunc returns the partition number as a value between 0 and (*num-partitions* - 1).

- This example uses the EMPLOYEE table from the sample database to show the partition number of each row based on the EMPNUM for four partitions and ordered by partition and EMPNUM.

  This example shows how HashPartFunc can reveal data skew. Because the EMPLOYEE table has only 62 different EMPNUMs, when it is partitioned four ways, the last partition is somewhat shorter than the others, because the number of unique entry counts (UECs) from the partitioning key is not sufficiently greater than the number of partitions.

```
>>SELECT empnum, HashPartFunc (empnum  for 4)
+>FROM employee
+>ORDER by 2, empnum;


Employee/Number  (EXPR)
---------------  ----------

             29           0
             43           0
             75           0
            109           0
            203           0
            208           0
            209           0
            219           0
            221           0
            224           0
            226           0
            229           0
            232           0
            235           0
            343           0
            557           0
            568           0
            991           0
             39           1
             89           1
             93           1
            201           1
            210           1
            217           1
            223           1
            225           1
            228           1
            230           1
            233           1
            321           1
            337           1
            990           1
            992           1
            994           1
            995           1
              1           2
```

```
                 23            2
                 32            2
                 65            2
                 87            2
                104            2
                178            2
                180            2
                202            2
                205            2
                207            2
                211            2
                212            2
                214            2
                215            2
                216            2
                218            2
                222            2
                227            2
                993            2
                 72            3
                111            3
                206            3
                213            3
                220            3
                231            3
                234            3

--- 62 row(s) selected.
```

- This example shows the number of rows that will reside in each partition if you create a new table that is hash-partitioned with four partitions, using the EMPNUM column from the EMPLOYEE table. This query is based on the same data results as the previous example, only grouped and ordered on the partition number.

```
   >>SELECT partitionNum, count(*)
   +>FROM (SELECT HashPartFunc (empnum  for 4)
   +>      FROM employee) as Tmp(partitionNum)
   +>GROUP BY partitionNum
   +>ORDER BY partitionNum;

   PARTITIONNUM  (EXPR)
   ------------  --------------------

              0                    18
              1                    17
              2                    20
              3                     7

--- 4 row(s) selected.
```

- This example shows the number of rows that will reside in each partition if you create a new table that is hash-partitioned with four partitions (using columns a and b as the partitioning key) and populate it with the 1000 data rows that currently reside in cat.sch.table1:

```
>>SELECT partitionNum, count(*)
+>FROM (SELECT HashPartFunc(CAST(a AS INT NOT NULL),
          CAST(b AS CHAR(3) NOT NULL) FOR 4)
+>        FROM cat.sch.table1) AS Tmp(partitionNum)
+>GROUP BY partitionNum
+>ORDER BY partitionNum;

PARTITIONNUM     (EXPR)
------------ --------------------

         0                   264
         1                   265
         2                   230
         3                   241

--- 4 row(s) selected.
```

- The HashPartFunc function supports null values. For example:

```
>>SELECT HashPartFunc(cast(null as INT) for 4) from
(values(0)) T;

   (EXPR)
   ------

       3

   --- 1 row(s) selected.
```

Normally, it is important to cast the values to the desired type, but in the case of null values, the type does not matter. Every null value hashes to the same value. However, there is no harm in keeping the cast for consistency. For example:

```
   >>SELECT HashPartFunc(cast(null as CHAR(10)) for 4) from
(values(0)) T;

   (EXPR)
   ------

       3

   --- 1 row(s) selected.

>>SELECT HashPartFunc(null for 4) from (values(0)) T;

   (EXPR)
   ------

       3

   --- 1 row(s) selected.
```

HOUR Function

The HOUR function converts a TIME or TIMESTAMP expression into an INTEGER value in the range 0 through 23 that represents the corresponding hour of the day.

HOUR is an SQL/MX extension.

```
HOUR (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type TIME or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of HOUR

- Return an integer that represents the hour of the day from the SHIP_TIMESTAMP column in the PROJECT table:

```
SELECT start_date, ship_timestamp, HOUR(ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
1996-04-10  1996-04-21 08:15:00.000000       8
```

INSERT Function

The INSERT function returns a character string where a specified number of characters within the character string have been deleted beginning at a specified start position and then another character string has been inserted at the start position.

INSERT is an SQL/MX extension.

```
INSERT (char-expr-1, start, length, char-expr-2)
```

char-expr-1, char-expr-2

are SQL character value expressions (of data type CHAR or VARCHAR) that specify two strings of characters. The character string *char-expr-2* is inserted into the character string *char-expr-1*. See Character Value Expressions on page 6-41.

start

specifies the starting position *start* within *char-expr-1* at which to start deleting *length* number of characters. After the deletion, the character string *char-expr-2* is inserted into the character string *char-expr-1*, beginning at the start position specified by the number *start*. The number *start* must be a value greater than zero of exact numeric data type and with a scale of zero.

length

specifies the number of characters to delete from *char-expr-1*. The number *length* must be a value greater than or equal to zero of exact numeric data type and with a scale of zero. *length* must be less than or equal to the length of *char-expr-1*.

Examples of INSERT

- Suppose that your JOB table includes an entry for a sales representative. Use the INSERT function to change SALESREP to SALES REP:

  ```
  UPDATE persnl.job
  SET jobdesc = INSERT (jobdesc, 6, 3, ' REP')
  WHERE jobdesc = 'SALESREP';
  ```

  Now check the row you updated:

  ```
  SELECT jobdesc FROM persnl.job
  WHERE jobdesc = 'SALES REP';

  Job Description
  ------------------
  SALES REP

  --- 1 row(s) selected.
  ```

JULIANTIMESTAMP Function

The JULIANTIMESTAMP function converts a datetime value into a 64-bit Julian timestamp value that represents the number of microseconds that have elapsed between 4713 B.C., January 1, 00:00, and the specified datetime value. JULIANTIMESTAMP returns a value of data type LARGEINT.

JULIANTIMESTAMP is an SQL/MX extension.

```
JULIANTIMESTAMP (datetime-expression)
```

datetime-expression

is an expression that evaluates to a value of type DATE, TIME, or TIMESTAMP. If *datetime-expression* does not contain all the fields from YEAR through SECOND, NonStop SQL/MX extends the value before converting it to a Julian timestamp. Datetime fields to the left of the specified datetime value are set to current date fields. Datetime fields to the right of the specified datetime value are set to zero. See Datetime Value Expressions on page 6-43.

Examples of JULIANTIMESTAMP

The PROJECT table consists of five columns using the data types NUMERIC, VARCHAR, DATE, TIMESTAMP, and INTERVAL.

- Convert the TIMESTAMP value into a Julian timestamp representation:

```
SELECT ship_timestamp, JULIANTIMESTAMP (ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;

SHIP_TIMESTAMP              (EXPR)
--------------------------  --------------------
1996-04-21 08:15:00.000000    211696834500000000

--- 1 row(s) selected.
```

- Convert the DATE value into a Julian timestamp representation:

```
SELECT start_date, JULIANTIMESTAMP (start_date)
FROM persnl.project
WHERE projcode = 1000;

START_DATE  (EXPR)
----------  --------------------
1996-04-10    211695854400000000

--- 1 row(s) selected.
```

LASTNOTNULL Function

The LASTNOTNULL function is a sequence function that returns the last non-null value of a column in an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

LASTNOTNULL is an SQL/MX extension.

```
LASTNOTNULL (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. If only null values have been returned, LASTNOTNULL returns null.

Examples of LASTNOTNULL

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
(I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data sequenced by TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
null	TIMESTAMP '1951-02-15 14:35:49'
19058	TIMESTAMP '1955-05-18 08:40:10'
null	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the last non-null value of a column:

```
SELECT LASTNOTNULL (I1) AS LASTNOTNULL
FROM mining.seqfcn SEQUENCE BY TS;

LASTNOTNULL
-----------
       6215
       6215
      19058
      19058
      11966

--- 5 row(s) selected.
```

LCASE Function

The LCASE function downshifts characters. LCASE can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the LCASE function is equal to the result returned by the LOWER function.

LCASE returns a string of either fixed-length or variable-length character data, depending on the data type of the input string.

You cannot use the LCASE function on KANJI or KSC5601 operands.

LCASE is an SQL/MX extension.

```
LCASE (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to downshift. See Character Value Expressions on page 6-41.

Examples of LCASE

- Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return in uppercase and lowercase letters by using the UCASE and LCASE functions:

```
SELECT custname,UCASE(custname),LCASE(custname)
FROM sales.customer;

(EXPR)              (EXPR)              (EXPR)
-----------------   -------------------   ------------------
...                 ...                   ...
Hotel Oregon        HOTEL OREGON          hotel oregon

--- 17 row(s) selected.
```

See UCASE Function on page 8-193.

LEFT Function

The LEFT function returns the leftmost specified number of characters from a character expression.

LEFT is an SQL/MX extension.

```
LEFT (character-expr, count)
```

character-expr

specifies the source string from which to return the leftmost specified number of characters. The source string is an SQL character value expression. The operand is the result of evaluating *character-expr*. See Character Value Expressions on page 6-41.

count

specifies the number of characters to return from *character-expr*. The number *count* must be a value of exact numeric data type greater than or equal to 0 with a scale of zero.

Examples of LEFT

- Return 'Robert':

```
LEFT ('Robert John Smith', 6)
```

- Use the LEFT function to append the company name to the job descriptions:

```
UPDATE persnl.job
SET jobdesc = LEFT (jobdesc, 11) ||' COMNET';

SELECT jobdesc FROM persnl.job;

Job Description
------------------

MANAGER COMNET
PRODUCTION  COMNET
ASSEMBLER COMNET
SALESREP COMNET
SYSTEM ANAL COMNET
ENGINEER COMNET
PROGRAMMER COMNET
ACCOUNTANT COMNET
ADMINISTRAT COMNET
SECRETARY COMNET

--- 10 row(s) selected.
```

LNNVL Function

The function takes `condition` as an argument, and when the `condition` is FALSE or NULL, it returns TRUE, or when the `condition` is TRUE, it returns FALSE. This function is only supported in the WHERE clause of the query.

```
LNNVL(condition)
```

The `condition` argument is a simple condition. The `condition` cannot be a compound condition containing AND, OR, or BETWEEN.

The function is logically equivalent to

```
if ((condition = FALSE) OR (condition = NULL))

    return TRUE;

else

    return FALSE;
```

Examples of LNNVL

```
create table tab1(colA int, colB int, colC int);

--- SQL operation complete.

insert into tab1(colA,colB) values(100,20);

-- 1 row(s) inserted.

insert into tab1 values(200,15,8);

--- 1 row(s) inserted.

insert into tab1 values(300,8,12);

-- 1 row(s) inserted.

insert into tab1 values(400,10,11);

--- 1 row(s) inserted.

insert into tab1 values(300,10,8);

--- 1 row(s) inserted.

insert into tab1 values(300,10,6);

-- 1 row(s) inserted.

Select * from tab1 where LNNVL(colB<=colC);
 COLA         COLB         COLC
-----------  -----------  -----------
        100           20            ?
        200           15            8
        300           10            8
        300           10            6
--- 4 row(s) selected.
```

- In this example, LNNVL (colB > colC), SELECT statement selects the records from the column where colB is not greater than colC including NULL values. The first record has a NULL value for colC, hence LNNVL function returns TRUE for that record.

```
Select * from tab1 where LNNVL(colB>colC);
COLA         COLB         COLC
----------  ----------- -----------
100          20            ?
300          8             12
400          0            11
--- 3 row(s) selected.
```

- The LNNVL (colC is NULL), SELECT statement selects the records where colC is NOT NULL.

```
Select * from tab1 where LNNVL(colC is NULL);

       COLA         COLB         COLC
-----------  -----------  -----------
        200           15            8
        300            8           12
        400           10           11
        300           10            8
        300           10            6
--- 5 row(s) selected.
```

LOCATE Function

The LOCATE function searches for a given substring in a character string. If the substring is found, NonStop SQL/MX returns the character position of the substring within the string. The result returned by the LOCATE function is equal to the result returned by the POSITION function.

LOCATE is an SQL/MX extension.

```
LOCATE (substring-expression,source-expression)
```

substring-expression

is an SQL character value expression that specifies the substring to search for in *source-expression*. The *substring-expression* cannot be NULL. See Character Value Expressions on page 6-41.

source-expression

is an SQL character value expression that specifies the source string. The *source-expression* cannot be NULL. See Character Value Expressions on page 6-41.

NonStop SQL/MX returns the result as a 2-byte signed integer with a scale of zero. If *substring-expression* is not found in *source-expression*, NonStop SQL/MX returns 0.

Considerations for LOCATE

Result of LOCATE

If the length of *source-expression* is zero and the length of *substring-expression* is greater than zero, NonStop SQL/MX returns 0. If the length of *substring-expression* is zero, NonStop SQL/MX returns 1.

If the length of *substring-expression* is greater than the length of *source-expression*, NonStop SQL/MX returns 0. If *source-expression* is a null value, NonStop SQL/MX returns a null value.

Using UCASE

To ignore the case in the search, use the UCASE function (or the LCASE function) for both the *substring-expression* and the *source-expression*.

Examples of LOCATE

- Return the value 8 for the position of the substring `‘John’` within the string:

  ```
  LOCATE ('John','Robert John Smith')
  ```

- Suppose that the EMPLOYEE table has an EMPNAME column that contains both the first and last names. This SELECT statement returns all records in table EMPLOYEE that contain the substring `'SMITH'`, regardless of whether the column value is in uppercase or lowercase characters:

  ```
  SELECT * FROM persnl.employee
    WHERE LOCATE ('SMITH',UCASE(empname)) > 0 ;
  ```

LOG Function

The LOG function returns the natural logarithm of a numeric value expression.

LOG is an SQL/MX extension.

```
LOG (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the LOG function. The value of the argument must be greater than zero. See Numeric Value Expressions on page 6-52.

Examples of LOG

- This function returns the value 6.93147180559945504E-001, or approximately 0.69315:

```
LOG (2.0)
```

LOG10 Function

The LOG10 function returns the base 10 logarithm of a numeric value expression.

LOG10 is an SQL/MX extension.

```
LOG10 (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the LOG10 function. The value of the argument must be greater than zero. See Numeric Value Expressions on page 6-52.

Examples of LOG10

- This function returns the value 1.39794000867203792E+000, or approximately 1.3979:

```
LOG10 (25)
```

LOWER Function

Considerations for LOWER
Examples of LOWER

The LOWER function downshifts characters. LOWER can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the LOWER function is equal to the result returned by the LCASE function.

For UCS2, the LOWER function downshifts all the uppercase or title case characters in a given string to lowercase and returns a character string with the same data type and character set as the argument.

A lower case character is a character that has the “alphabetic” property in Unicode Standard 2 whose Unicode name includes *lower*. An uppercase character is a character that has the “alphabetic” property in the Unicode Standard 2 and whose Unicode name includes *upper*. A title case character is a character that has the Unicode “alphabetic” property and whose Unicode name includes *title*.

You cannot use the LOWER function on KANJI or KSC5601 operands.

LOWER returns a string of either fixed-length or variable-length character data, depending on the data type of the input string.

```
LOWER (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to downshift. See Character Value Expressions on page 6-41.

Considerations for LOWER

Table 8-2 lists all one-to-one mappings for the UCS2 character set.

Table 8-2. One-to-One Uppercase and Titlecase to Lowercase Mappings (page 1 of 4)

x	L (x)	x	L(x)	x	L(x)	x	L(x)	x	L(x)	x	L(x)
0041	0061	017B	017C	03EC	03ED	0536	0566	1E5C	1E5D	1F6B	1F63
0042	0062	017D	017E	03EE	03EF	0537	0567	1E5E	1E5F	1F6C	1F64
0043	0063	0181	0253	0401	0451	0538	0568	1E60	1E61	1F6D	1F65
0044	0064	0182	0183	0402	0452	0539	0569	1E62	1E63	1F6E	1F66
0045	0065	0184	0185	0403	0453	053A	056A	1E64	1E65	1F6F	1F67
0046	0066	0186	0254	0404	0454	053B	056B	1E66	1E67	1F88	1F80
0047	0067	0187	0188	0405	0455	053C	056C	1E68	1E69	1F89	1F81
0048	0068	0189	0256	0406	0456	053D	056D	1E6A	1E6B	1F8A	1F82

Table 8-2. One-to-One Uppercase and Titlecase to Lowercase Mappings (page 2 of 4)

x	L (x)	x	L(x)	x	L(x)	x	L(x)	x	L(x)	x	L(x)
0049	0069	018A	0257	0407	0457	053E	056E	1E6C	1E6D	1F8B	1F83
004A	006A	018B	018C	0408	0458	053F	056F	1E6E	1E6F	1F8C	1F84
004B	006B	018E	01DD	0409	0459	0540	0570	1E70	1E71	1F8D	1F85
004C	006C	018F	0259	040A	045A	0541	0571	1E72	1E73	1F8E	1F86
004D	006D	0190	025B	040B	045B	0542	0572	1E74	1E75	1F8F	1F87
004E	006E	0191	0192	040C	045C	0543	0573	1E76	1E77	1F98	1F90
004F	006F	0193	0260	040E	045E	0544	0574	1E78	1E79	1F99	1F91
0050	0070	0194	0263	040F	045F	0545	0575	1E7A	1E7B	1F9A	1F92
0051	0071	0196	0269	0410	0430	0546	0576	1E7C	1E7D	1F9B	1F93
0052	0072	0197	0268	0411	0431	0547	0577	1E7E	1E7F	1F9C	1F94
0053	0073	0198	0199	0412	0432	0548	0578	1E80	1E81	1F9D	1F95
0054	0074	019C	026F	0413	0433	0549	0579	1E82	1E83	1F9E	1F96
0055	0075	019D	0272	0414	0434	054A	057A	1E84	1E85	1F9F	1F97
0056	0076	019F	0275	0415	0435	054B	057B	1E86	1E87	1FA8	1FA0
0057	0077	01A0	01A1	0416	0436	054C	057C	1E88	1E89	1FA9	1FA1
0058	0078	01A2	01A3	0417	0437	054D	057D	1E8A	1E8B	1FAA	1FA2
0059	0079	01A4	01A5	0418	0438	054E	057E	1E8C	1E8D	1FAB	1FA3
005A	007A	01A6	0280	0419	0439	054F	057F	1E8E	1E8F	1FAC	1FA4
00C0	00E0	01A7	01A8	041A	043A	0550	0580	1E90	1E91	1FAD	1FA5
00C1	00E1	01A9	0283	041B	043B	0551	0581	1E92	1E93	1FAE	1FA6
00C2	00E2	01AC	01AD	041C	043C	0552	0582	1E94	1E95	1FAF	1FA7
00C3	00E3	01AE	0288	041D	043D	0553	0583	1EA0	1EA1	1FB8	1FB0
00C4	00E4	01AF	01B0	041E	043E	0554	0584	1EA2	1EA3	1FB9	1FB1
00C5	00E5	01B1	028A	041F	043F	0555	0585	1EA4	1EA5	1FBA	1F70
00C6	00E6	01B2	028B	0420	0440	0556	0586	1EA6	1EA7	1FBB	1F71
00C7	00E7	01B3	01B4	0421	0441	10A0	10D0	1EA8	1EA9	1FBC	1FB3
00C8	00E8	01B5	01B6	0422	0442	10A1	10D1	1EAA	1EAB	1FC8	1F72
00C9	00E9	01B7	0292	0423	0443	10A2	10D2	1EAC	1EAD	1FC9	1F73
00CA	00EA	01B8	01B9	0424	0444	10A3	10D3	1EAE	1EAF	1FCA	1F74
00CB	00EB	01BC	01BD	0425	0445	10A4	10D4	1EB0	1EB1	1FCB	1F75
00CC	00EC	01C4	01C6	0426	0446	10A5	10D5	1EB2	1EB3	1FCC	1FC3
00CD	00ED	01C5	01C6	0427	0447	10A6	10D6	1EB4	1EB5	1FD8	1FD0
00CE	00EE	01C7	01C9	0428	0448	10A7	10D7	1EB6	1EB7	1FD9	1FD1
00CF	00EF	01C8	01C9	0429	0449	10A8	10D8	1EB8	1EB9	1FDA	1F76
00D0	00F0	01CA	01CC	042A	044A	10A9	10D9	1EBA	1EBB	1FDB	1F77
00D1	00F1	01CB	01CC	042B	044B	10AA	10DA	1EBC	1EBD	1FE8	1FE0
00D2	00F2	01CD	01CE	042C	044C	10AB	10DB	1EBE	1EBF	1FE9	1FE1
00D3	00F3	01CF	01D0	042D	044D	10AC	10DC	1EC0	1EC1	1FEA	1F7A

Table 8-2. One-to-One Uppercase and Titlecase to Lowercase Mappings (page 3 of 4)

x	L (x)	x	L(x)	x	L(x)	x	L(x)	x	L(x)	x	L(x)
00D4	00F4	01D1	01D2	042E	044E	10AD	10DD	1EC2	1EC3	1FEB	1F7B
00D5	00F5	01D3	01D4	042F	044F	10AE	10DE	1EC4	1EC5	1FEC	1FE5
00D6	00F6	01D5	01D6	0460	0461	10AF	10DF	1EC6	1EC7	1FF8	1F78
00D8	00F8	01D7	01D8	0462	0463	10B0	10E0	1EC8	1EC9	1FF9	1F79
00D9	00F9	01D9	01DA	0464	0465	10B1	10E1	1ECA	1ECB	1FFA	1F7C
00DA	00FA	01DB	01DC	0466	0467	10B2	10E2	1ECC	1ECD	1FFB	1F7D
00DB	00FB	01DE	01DF	0468	0469	10B3	10E3	1ECE	1ECF	1FFC	1FF3
00DC	00FC	01E0	01E1	046A	046B	10B4	10E4	1ED0	1ED1	2160	2170
00DD	00FD	01E2	01E3	046C	046D	10B5	10E5	1ED2	1ED3	2161	2171
00DE	00FE	01E4	01E5	046E	046F	10B6	10E6	1ED4	1ED5	2162	2172
0100	0101	01E6	01E7	0470	0471	10B7	10E7	1ED6	1ED7	2163	2173
0102	0103	01E8	01E9	0472	0473	10B8	10E8	1ED8	1ED9	2164	2174
0104	0105	01EA	01EB	0474	0475	10B9	10E9	1EDA	1EDB	2165	2175
0106	0107	01EC	01ED	0476	0477	10BA	10EA	1EDC	1EDD	2166	2176
0108	0109	01EE	01EF	0478	0479	10BB	10EB	1EDE	1EDF	2167	2177
010A	010B	01F1	01F3	047A	047B	10BC	10EC	1EE0	1EE1	2168	2178
010C	010D	01F2	01F3	047C	047D	10BD	10ED	1EE2	1EE3	2169	2179
010E	010F	01F4	01F5	047E	047F	10BE	10EE	1EE4	1EE5	216A	217A
0110	0111	01FA	01FB	0480	0481	10BF	10EF	1EE6	1EE7	216B	217B
0112	0113	01FC	01FD	0490	0491	10C0	10F0	1EE8	1EE9	216C	217C
0114	0115	01FE	01FF	0492	0493	10C1	10F1	1EEA	1EEB	216D	217D
0116	0117	0200	0201	0494	0495	10C2	10F2	1EEC	1EED	216E	217E
0118	0119	0202	0203	0496	0497	10C3	10F3	1EEE	1EEF	216F	217F
011A	011B	0204	0205	0498	0499	10C4	10F4	1EF0	1EF1	24B6	24D0
011C	011D	0206	0207	049A	049B	10C5	10F5	1EF2	1EF3	24B7	24D1
011E	011F	0208	0209	049C	049D	1E00	1E01	1EF4	1EF5	24B8	24D2
0120	0121	020A	020B	049E	049F	1E02	1E03	1EF6	1EF7	24B9	24D3
0122	0123	020C	020D	04A0	04A1	1E04	1E05	1EF8	1EF9	24BA	24D4
0124	0125	020E	020F	04A2	04A3	1E06	1E07	1F08	1F00	24BB	24D5
0126	0127	0210	0211	04A4	04A5	1E08	1E09	1F09	1F01	24BC	24D6
0128	0129	0212	0213	04A6	04A7	1E0A	1E0B	1F0A	1F02	24BD	24D7
012A	012B	0214	0215	04A8	04A9	1E0C	1E0D	1F0B	1F03	24BE	24D8
012C	012D	0216	0217	04AA	04AB	1E0E	1E0F	1F0C	1F04	24BF	24D9
012E	012F	0386	03AC	04AC	04AD	1E10	1E11	1F0D	1F05	24C0	24DA
0130	0069	0388	03AD	04AE	04AF	1E12	1E13	1F0E	1F06	24C1	24DB
0132	0133	0389	03AE	04B0	04B1	1E14	1E15	1F0F	1F07	24C2	24DC
0134	0135	038A	03AF	04B2	04B3	1E16	1E17	1F18	1F10	24C3	24DD
0136	0137	038C	03CC	04B4	04B5	1E18	1E19	1F19	1F11	24C4	24DE

Table 8-2. One-to-One Uppercase and Titlecase to Lowercase Mappings (page 4 of 4)

x	L (x)	x	L(x)	x	L(x)	x	L(x)	x	L(x)	x	L(x)
0139	013A	038E	03CD	04B6	04B7	1E1A	1E1B	1F1A	1F12	24C5	24DF
013B	013C	038F	03CE	04B8	04B9	1E1C	1E1D	1F1B	1F13	24C6	24E0
013D	013E	0391	03B1	04BA	04BB	1E1E	1E1F	1F1C	1F14	24C7	24E1
013F	0140	0392	03B2	04BC	04BD	1E20	1E21	1F1D	1F15	24C8	24E2
0141	0142	0393	03B3	04BE	04BF	1E22	1E23	1F28	1F20	24C9	24E3
0143	0144	0394	03B4	04C1	04C2	1E24	1E25	1F29	1F21	24CA	24E4
0145	0146	0395	03B5	04C3	04C4	1E26	1E27	1F2A	1F22	24CB	24E5
0147	0148	0396	03B6	04C7	04C8	1E28	1E29	1F2B	1F23	24CC	24E6
014A	014B	0397	03B7	04CB	04CC	1E2A	1E2B	1F2C	1F24	24CD	24E7
014C	014D	0398	03B8	04D0	04D1	1E2C	1E2D	1F2D	1F25	24CE	24E8
014E	014F	0399	03B9	04D2	04D3	1E2E	1E2F	1F2E	1F26	24CF	24E9
0150	0151	039A	03BA	04D4	04D5	1E30	1E31	1F2F	1F27	FF21	FF41
0152	0153	039B	03BB	04D6	04D7	1E32	1E33	1F38	1F30	FF22	FF42
0154	0155	039C	03BC	04D8	04D9	1E34	1E35	1F39	1F31	FF23	FF43
0156	0157	039D	03BD	04DA	04DB	1E36	1E37	1F3A	1F32	FF24	FF44
0158	0159	039E	03BE	04DC	04DD	1E38	1E39	1F3B	1F33	FF25	FF45
015A	015B	039F	03BF	04DE	04DF	1E3A	1E3B	1F3C	1F34	FF26	FF46
015C	015D	03A0	03C0	04E0	04E1	1E3C	1E3D	1F3D	1F35	FF27	FF47
015E	015F	03A1	03C1	04E2	04E3	1E3E	1E3F	1F3E	1F36	FF28	FF48
0160	0161	03A3	03C3	04E4	04E5	1E40	1E41	1F3F	1F37	FF29	FF49
0162	0163	03A4	03C4	04E6	04E7	1E42	1E43	1F48	1F40	FF2A	FF4A
0164	0165	03A5	03C5	04E8	04E9	1E44	1E45	1F49	1F41	FF2B	FF4B
0166	0167	03A6	03C6	04EA	04EB	1E46	1E47	1F4A	1F42	FF2C	FF4C
0168	0169	03A7	03C7	04EE	04EF	1E48	1E49	1F4B	1F43	FF2D	FF4D
016A	016B	03A8	03C8	04F0	04F1	1E4A	1E4B	1F4C	1F44	FF2E	FF4E
016C	016D	03A9	03C9	04F2	04F3	1E4C	1E4D	1F4D	1F45	FF2F	FF4F
016E	016F	03AA	03CA	04F4	04F5	1E4E	1E4F	1F59	1F51	FF30	FF50
0170	0171	03AB	03CB	04F8	04F9	1E50	1E51	1F5B	1F53	FF31	FF51
0172	0173	03E2	03E3	0531	0561	1E52	1E53	1F5D	1F55	FF32	FF52
0174	0175	03E4	03E5	0532	0562	1E54	1E55	1F5F	1F57	FF33	FF53
0176	0177	03E6	03E7	0533	0563	1E56	1E57	1F68	1F60	FF34	FF54
0178	00FF	03E8	03E9	0534	0564	1E58	1E59	1F69	1F61	FF35	FF55
0179	017A	03EA	03EB	0535	0565	1E5A	1E5B	1F6A	1F62	FF36	FF56
										FF37	FF57
										FF38	FF58
										FF39	FF59
										FF3A	FF5A

Examples of LOWER

- Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return the result in uppercase and lowercase letters by using the UPPER and LOWER functions:

```
SELECT custname,UPPER(custname),LOWER(custname)
FROM sales.customer;

(EXPR)              (EXPR)               (EXPR)
-----------------   -------------------  ------------------
...                 ...                  ...
Hotel Oregon        HOTEL OREGON         hotel oregon

--- 17 row(s) selected.
```

See UPPER Function on page 8-201.

LPAD Function

The LPAD function replaces the leftmost specified number of characters in a character expression with a padding character or string. With the ANSI_STRING_FUNCTIONALITY CQD set to ON, the function pads the left side of a character expression with the specified string.

```
LPAD (character-expr,count [,pad-character])
```

character-expr

is an SQL character value expression. The operand is the result of evaluating *character-expr*. See Character Value Expressions on page 6-41.

count

specifies the number of characters. The *count* must be greater than or equal to zero of exact numeric data type and with a scale of zero. For considerations of *count* based on CQD ANSI_STRING_FUNCTIONALITY, see Examples of LPAD on page 8-99.

pad-character

specifies the padding character or a string. If no *pad-character* is specified, space is the padding character. For KANJI or KSC5601, the code value of *pad-character* is hexadecimal 2020.

Examples of LPAD

The behavior of the LPAD function when the ANSI_STRING_FUNCTIONALITY CQD is set to ON and the corresponding examples are described below.

The *count* specifies the number of characters to be returned. It is the length of the result string.

If *count* is smaller than the length of the *character-expr*, the *character-expr* is truncated. If *count* is equal to the length of the *character-expr*, the value of the *character-expr* is retained. If *count* is greater than the length of the *character-expr*, the *character-expr* is left-padded with the pad-character.

- The following LPAD function truncates the string 'kite' and returns two leftmost characters 'ki':

  ```
  LPAD('kite', 2)
  ```

- The following LPAD function truncates the string 'Robert John Smith' and returns six leftmost characters 'Robert':

  ```
  LPAD('Robert John Smith', 6);
  ```

- The following LPAD function returns the original string '`go fly a kite`' because the *`count`* is equal to the length of the string:

  ```
  LPAD('go fly a kite', 13, 'z')
  ```

- The following LPAD function returns a string of 10 characters '`    Robert`' by left padding the string '`Robert`' with four spaces:

  ```
  LPAD('Robert',10)
  ```

- The following LPAD function returns a string of eight characters '`0000kite`' by left padding the string '`kite`' with four pad-characters '`0`':

  ```
  LPAD('kite', 8, '0')
  ```

- The following LPAD function returns '`John,John, go fly a kite`':

  ```
  LPAD('go fly a kite', 23, 'John,')
  ```

  The function left pads the string '`go fly a kite`' with the string '`John,`' such that the length of the result string is 23 characters.

- The following LPAD function returns '`John,Jogo fly a kite`':

  ```
  LPAD('go fly a kite', 20, 'John,')
  ```

  The function left pads the string '`go fly a kite`' with the string '`John,`' such that the length of result string is 20 characters.

The default behavior of the LPAD function and the corresponding examples are described below.

The *`count`* specifies the number of characters to be replaced. The *`count`* must be less than or equal to the length of the *`character-expr`*. If *`count`* is smaller than or equal to the length of the *`character-expr`*, the leftmost *`count`* characters of the *`character-expr`* are replaced with the padding characters or string. If *`count`* is greater than the length of the *`character-expr`*, an error is returned.

- The following LPAD function replaces two leftmost characters in the string '`kite`' with spaces and returns '`  te`':

  ```
  LPAD('kite', 2)
  ```

- The following LPAD function replaces six leftmost characters in the string '`Robert John Smith`' with spaces and returns '`       John Smith`':

  ```
  LPAD('Robert John Smith', 6);
  ```

- The following LPAD function replaces two leftmost characters 'go' with the string '`John,`' twice and returns '`John,John, fly a kite`':

  ```
  LPAD('go fly a kite', 2, 'John,')
  ```

- The following LPAD function replaces 13 leftmost characters in the string '`go fly a kite`' with character 'z' and returns '`zzzzzzzzzzzzz`'

  ```
  LPAD('go fly a kite', 13, 'z')
  ```

- The following LPAD functions return an error because the *`count`* is greater than the string length:

  ```
  LPAD('Robert',10)
  LPAD('kite', 8, '0')
  LPAD('go fly a kite', 23, 'John,')
  ```

LTRIM Function

The LTRIM function removes the specified trim characters from the left side of the character string. If the trim characters are not specified, by default, the function removes spaces.

```
LTRIM(srcstr[, trim_chrs])
```

srcstr

is a SQL character value expression from which spaces or *trim_chrs* are removed from the left side. See Character Value Expressions on page 6-41.

trim_chrs

is the character or characters to be removed from the *srcstr* . The default is the space character.

Considerations for LTRIM

Result of LTRIM

The result is always of type NON ANSI VARCHAR, with maximum length equal to the fixed length or maximum variable length of *srcstr*.

Examples of LTRIM

- Return `'Robert    '`:

  ```
  LTRIM ('    Robert    ')
  ```

  See TRIM Function on page 8-192 and RTRIM Function on page 8-152.

- In this example, characters '1' and '0' are removed from the left side of the `colA` value:

  ```
  create table tab1(colA varchar(20), colB int);
  insert into tab1 values('0101010101ten', 10);
  insert into tab1 values('11001101nine', 9);
  select ltrim(colA, '10'), colB from tab1;
  (EXPR)                COLB
  --------------------  -----------
  ten                            10
  nine                            9
  ```

MAX Function

MAX is an aggregate function that returns the maximum value within a set of values. The data type of the result is the same as the data type of the argument.

```
MAX ([ALL | DISTINCT] expression)
```

`ALL | DISTINCT`

specifies whether duplicate values are included in the computation of the maximum of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the MAX function is applied.

expression

specifies an expression that determines the values to include in the computation of the maximum. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the MAX function operates on distinct values from the one-column table derived from the evaluation of *expression*. All nulls are eliminated before the function is applied to the set of values. If the result table is empty, MAX returns NULL.

See Expressions on page 6-41.

Considerations for MAX

Operands of the Expression

The expression includes columns from the rows of the SELECT result table but cannot include an aggregate function. These expressions are valid:

```
MAX (SALARY)
MAX (SALARY * 1.1)
MAX (PARTCOST * QTY_ORDERED)
```

Examples of MAX

- Display the maximum value in the SALARY column:

```
SELECT MAX (salary)
FROM persnl.employee;

(EXPR)
-----------
  175500.00

--- 1 row(s) selected.
```

MIN Function

MIN is an aggregate function that returns the minimum value within a set of values. The data type of the result is the same as the data type of the argument.

```
MIN ([ALL | DISTINCT] expression)
```

`ALL | DISTINCT`

specifies whether duplicate values are included in the computation of the minimum of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the MIN function is applied.

expression

specifies an expression that determines the values to include in the computation of the minimum. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the MIN function operates on distinct values from the one-column table derived from the evaluation of *expression*. All nulls are eliminated before the function is applied to the set of values. If the result table is empty, MIN returns NULL.

See Expressions on page 6-41.

Considerations for MIN

Operands of the Expression

The expression includes columns from the rows of the SELECT result table—but cannot include an aggregate function. These expressions are valid:

```
MIN (SALARY)
MIN (SALARY * 1.1)
MIN (PARTCOST * QTY_ORDERED)
```

Examples of MIN

- Display the minimum value in the SALARY column:

```
SELECT MIN (salary)
FROM persnl.employee;

(EXPR)
-----------
   17000.00

--- 1 row(s) selected.
```

MINUTE Function

The MINUTE function converts a TIME or TIMESTAMP expression into an INTEGER value, in the range 0 through 59, that represents the corresponding minute of the hour.

MINUTE is an SQL/MX extension.

```
MINUTE (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type TIME or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of MINUTE

- Return an integer that represents the minute of the hour from the SHIP_TIMESTAMP column in the PROJECT table:

```
SELECT start_date, ship_timestamp, MINUTE(ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
1996-04-10  1996-04-21 08:15:00.000000      15
```

MOD Function

The MOD function returns the remainder (modulus) of an integer value expression divided by an integer value expression.

MOD is an SQL/MX extension.

```
MOD (integer-expression-1,integer-expression-2)
```

integer-expression-1

is an SQL numeric value expression of data type SMALLINT, INTEGER, or LARGEINT that specifies the value for the dividend argument of the MOD function.

integer-expression-2

is an SQL numeric value expression of data type SMALLINT, INTEGER, or LARGEINT that specifies the value for the divisor argument of the MOD function. The divisor argument cannot be zero.

See Numeric Value Expressions on page 6-52.

Examples of MOD

- This function returns the value 2 as the remainder or modulus:

  ```
  MOD (11,3)
  ```

MONTH Function

The MONTH function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 12 that represents the corresponding month of the year.

MONTH is an SQL/MX extension.

```
MONTH (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of MONTH

- Return an integer that represents the month of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, MONTH(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
1996-04-10  1996-04-21 08:15:00.000000       4
```

MONTHNAME Function

The MONTHNAME function converts a DATE or TIMESTAMP expression into a character literal that is the name of the month of the year (January, February, and so on).

MONTHNAME is an SQL/MX extension.

```
MONTHNAME (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of MONTHNAME

- Return a character literal that is the month of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, MONTHNAME(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ---------
1996-04-10  1996-04-21 08:15:00.000000  April
```

MOVINGAVG Function

The MOVINGAVG function is a sequence function that returns the average of non-null values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

MOVINGAVG is an SQL/MX extension.

```
MOVINGAVG (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGAVG returns the same result as RUNNINGAVG:
    - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
    - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range, if it is larger than the size of the result, larger than DEF_MAX_HISTORY_ ROWS table, negative, or NULL.

Examples of MOVINGAVG

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the average of non-null values of a column in the current window of three rows:

  ```
  SELECT MOVINGAVG (I1,3) AS MOVINGAVG3 FROM mining.seqfcn SEQUENCE BY TS;

  MOVINGAVG3
  --------------------

   6215
  17194
  17194
  16385
   8281

  --- 5 row(s) selected
  ```

Note.

1. The size the history buffer must not equal to the total result of the query. The required size of the history buffer might be the largest window size for computing a MOVINGXXX function, an OFFSET function or a ROWS SICE function.
2. Examining a large history buffer for a false condition will be an expensive operation. Such operation cannot be parallelized, for example, the ROWS SINCE function.

MOVINGCOUNT Function

The MOVINGCOUNT function is a sequence function that returns the number of non-null values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

MOVINGCOUNT is an SQL/MX extension.

```
MOVINGCOUNT (column-expression,integer-expression
            [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGCOUNT returns the same result as RUNNINGCOUNT:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result, larger then table, negative, or NULL.

Considerations for MOVINGCOUNT

No DISTINCT Clause

The MOVINGCOUNT sequence function is defined differently from the COUNT aggregate function. If you specify DISTINCT for the COUNT aggregate function, duplicate values are eliminated before COUNT is applied. Note that you cannot specify DISTINCT for the MOVINGCOUNT sequence function; duplicate values are counted.

Examples of MOVINGCOUNT

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the number of non-null values of a column in the current window of three rows:

```
SELECT MOVINGCOUNT (I1,3) AS MOVINGCOUNT3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGCOUNT3
------------
           1
           2
           2
           2
           2

--- 5 row(s) selected.
```

MOVINGMAX Function

The MOVINGMAX function is a sequence function that returns the maximum of non-null values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

MOVINGMAX is an SQL/MX extension.

```
MOVINGMAX (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* `- 1`) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGMAX returns the same result as RUNNINGMAX:
    - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
    - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Examples of MOVINGMAX

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	**TS**
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the maximum of non-null values of a column in the current window of three rows:

  ```
  SELECT MOVINGMAX (I1,3) AS MOVINGMAX3
  FROM mining.seqfcn
  SEQUENCE BY TS;

  MOVINGMAX3
  ------------
         6215
        28174
        28174
        28174
        11966

  --- 5 row(s) selected.
  ```

MOVINGMIN Function

The MOVINGMIN function is a sequence function that returns the minimum of non-null values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

MOVINGMIN is an SQL/MX extension.

```
MOVINGMIN (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (`integer-expression - 1`) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGMIN returns the same result as RUNNINGMIN:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Examples of MOVINGMIN

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	**TS**
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the minimum of non-null values of a column in the current window of three rows:

```
SELECT MOVINGMIN (I1,3) AS MOVINGMIN3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGMIN3
------------
        6215
        6215
        6215
        4597
        4597

--- 5 row(s) selected.
```

MOVINGSTDDEV Function

The MOVINGSTDDEV function is a sequence function that returns the standard deviation of non-null values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

MOVINGSTDDEV is an SQL/MX extension.

```
MOVINGSTDDEV (column-expression,integer-expression
              [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGSTDDEV returns the same result as RUNNINGSTDDEV:
    - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
    - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Examples of MOVINGSTDDEV

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	**TS**
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the standard deviation of non-null values of a column in the current window of three rows:

  ```
  SELECT MOVINGSTDDEV (I1,3) AS MOVINGSTDDEV3
  FROM mining.seqfcn
  SEQUENCE BY TS;

  MOVINGSTDDEV3
  -------------------------
   0.00000000000000000E+000
   1.55273578080753976E+004
   1.48020166531456112E+004
   1.51150124820766640E+004
   6.03627542446499008E+003

  --- 5 row(s) selected.
  ```

  Note that you can use the CAST function for display purposes. For example:

  ```
  SELECT CAST(MOVINGSTDDEV (I1,3) AS DEC (18,3))
  FROM mining.seqfcn
  SEQUENCE BY TS;

  (EXPR)
  --------------------
                  .000
             15527.357
             14802.016
             15115.012
              6036.275

  --- 5 row(s) selected.
  ```

MOVINGSUM Function

The MOVINGSUM function is a sequence function that returns the sum of non-null values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

MOVINGSUM is an SQL/MX extension.

```
MOVINGSUM (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGSUM returns the same result as RUNNINGSUM:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Examples of MOVINGSUM

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	**TS**
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the sum of non-null values of a column in the current window of three rows:

```
SELECT MOVINGSUM (I1,3) AS MOVINGSUM3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGSUM3
------------
        6215
       34389
       34389
       32771
       16563

--- 5 row(s) selected.
```

MOVINGVARIANCE Function

- The MOVINGVARIANCE function is a sequence function that returns the variance of non-null values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-18.
- MOVINGVARIANCE is an SQL/MX extension.
- MOVINGVARIANCE (column-expression,integer-expression
- [,max-rows])
-
- column-expression
- specifies a derived column determined by the evaluation of the column expression.
- integer-expression
- is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (integer-expression - 1) rows.
- max-rows
- is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.
- Note these considerations for the window size: The actual value for the window size is the minimum of `integer-expression` and `max-rows`.
- If these conditions are met, MOVINGVARIANCE returns the same result as RUNNINGVARIANCE:
    - The `integer-expression` is out of range, and `max-rows` is not specified. This condition includes the case in which both `integer-expression` and `max-rows` are larger than the result table.
    - The minimum of `integer-expression` and `max-rows` is out of range. In this case, `integer-expression` could be within range, but `max-rows` might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Examples of MOVINGVARIANCE

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the variance of non-null values of a column in the current window of three rows:

```
SELECT MOVINGVARIANCE (I1,3) AS MOVINGVARIANCE3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGVARIANCE3
-------------------------
 0.00000000000000000E+000
 2.41098840499999960E+008
 2.19099696999999968E+008
 2.28463602333333304E+008
 3.64366210000000016E+007

--- 5 row(s) selected.
```

Note that you can use the CAST function for display purposes. For example:

```
SELECT CAST(MOVINGVARIANCE (I1,3) AS DEC (18,3))
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)
--------------------
                .000
       241098840.500
       219099697.000
       228463602.333
        36436621.000

--- 5 row(s) selected.
```

NVL Function

The NVL function returns `expr2` if `expr1` is NULL. Otherwise, the function returns `expr1`.

```
NVL(expr1,expr2)
```

expr1 and *expr2* specify value expressions and must be of comparable data types.

The NVL function is logically equivalent to the following IF-ELSE expression:

```
if (expr1 is NULL)
 return expr2
else
 return expr1
```

Considerations

- If all the expressions in the argument list are fixed-length character types, the return value is a fixed-length character string equal to the maximum size of all the expression value types.
- If any of the expressions in the argument list are variable-length character types, the return value is a variable-length character string with maximum size of all the expression value types.
- If all the expressions in the argument list are integer types, the return value is of the same data type as the largest integer type of all the possible return values.
- If all the expressions in the argument list are numeric types, and at least one is REAL, FLOAT, or DOUBLE PRECISION, the return value is a DOUBLE PRECISION type.
- If all the expressions in the argument list are numeric types, and none are REAL, FLOAT, or DOUBLE PRECISION, and at least one is of type NUMERIC, the return value is a NUMERIC type.
- If the `expr1` is not NULL, the return value is the same data type as `expr1`.
- If the return value is of type NUMERIC or DECIMAL, it has a precision value equal to the sum of:
    - The maximum scale of all the return values.
    - The maximum of precision minus scale value for all the return values.

      However, the precision value must not exceed 18.
- If the return value is of type NUMERIC or DECIMAL, the scale of the return value is the minimum of:
    - The maximum scale of all the return values.
    - 18 - (the maximum of precision minus scale value for all the return values).

Note. Existing NonStop SQL/MX conversions rules are applied to the above rules**.**

Example of NVL

- This function replaces all NULL values with ‘`two`’.

```
create table tab1 (colA varchar(10), colB int);
insert into tab1 values('one',1);

-- The default value for colA is NULL.
insert into tab1(colB) values(2);

insert into tab1 values('three',3);
insert into tab1 values('four',4);

select colA, colB, NVL(colA, 'two') from tab1;

COLA        COLB         (EXPR)
----------  -----------  ----------

one         1            one
?           2            two
three       3            three
four        4            four
--- 4 row(s) selected.
```

NVL2 Function

The NVL2 function returns *expr2* if *expr1* is NOT NULL and returns *expr3* if *expr1* is NULL.

```
NVL2 (expr1, expr2, expr3)
```

expr1, expr2, expr3... are SQL value expressions.

The NVL2 function is logically equivalent to the following IF-ELSE expression:

```
if (expr1 is NOT NULL)
 return expr2
else
 return expr3
```

Considerations

The considerations of NVL2 are same as Considerations in NVL.

Examples of NVL2

- This function returns `colB` if `colA` is NOT NULL else returns integer ‘0’(third argument of NVL2).

```
create table tab1(colA varchar(10), colB int);
--- SQL operation complete.

insert into tab1 values('one', 1);
--- 1 row(s) inserted.

-- Default values for colA is NULL
insert into tab1(colB) values(2);
--- 1 row(s) inserted.
```

```
insert into tab1(colB) values(3);
--- 1 row(s) inserted.

insert into tab1 values('four',4);
--- 1 row(s) inserted.

select NVL2(colA,colB,0) as nvl2_value from tab1;

NVL2_VALUE
---------------------
1
0
0
4
```

OCTET_LENGTH Function

The OCTET_LENGTH function returns the length of a character string in bytes.

```
OCTET_LENGTH (string-value-expression)
```

string-value-expression

specifies the string value expression for which to return the length in bytes. NonStop SQL/MX returns the result as a 2-byte signed integer with a scale of zero. If *string-value-expression* is null, NonStop SQL/MX returns a length of zero. See Character Value Expressions on page 6-41.

Considerations for OCTET_LENGTH

CHAR and VARCHAR Operands

For a column declared as fixed CHAR, NonStop SQL/MX returns the length of that column as the maximum number of storage bytes. For a VARCHAR column, NonStop SQL/MX returns the length of the string stored in that column as the actual number of storage bytes.

Similarity to CHAR_LENGTH Function

The OCTET_LENGTH and CHAR_LENGTH functions are similar. The OCTET_LENGTH function returns the number of bytes, rather than the number of characters, in the string. This distinction is important for multibyte implementations. For an example of selecting a double-byte column, see Similarity to OCTET_LENGTH Function on page 8-24.

Examples of OCTET_LENGTH

- If a character string is stored as two bytes for each character, this function returns the value 12. Otherwise, the function returns 6:

  ```
  OCTET_LENGTH ('Robert')
  ```

OFFSET Function

The OFFSET function is a sequence function that retrieves columns from previous rows of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

OFFSET is an SQL/MX extension.

```
OFFSET (column-expression,number-rows [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

number-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the offset as the number of rows from the current row. If the number of rows exceeds *max-rows*, OFFSET returns OFFSET(*column-expression*,*max-rows*). If the number of rows is out of range and *max-rows* is not specified or is out of range, OFFSET returns null. The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows of the offset.

Examples of OFFSET

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
19058	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Retrieve the I1 column offset by three rows:

```
SELECT OFFSET (I1,3) AS OFFSET3
FROM mining.seqfcn
SEQUENCE BY TS;

OFFSET3
------------
           ?
           ?
           ?
        6215
       28174

--- 5 row(s) selected.
```

Note that the first three rows retrieved display null because the offset from the current row does not fall within the result table.

PI Function

The PI function returns the constant value of pi as a floating-point value.

PI is an SQL/MX extension.

```
PI()
```

Examples of PI

- This constant function returns the value 3.14159260000000064E+000:

  ```
  PI()
  ```

POSITION Function

The POSITION function searches for a given substring in a character string. If the substring is found, NonStop SQL/MX returns the character position of the substring within the string. The result returned by the POSITION function is equal to the result returned by the LOCATE function.

```
POSITION (substring-expression IN source-expression)
```

substring-expression

is an SQL character value expression that specifies the substring to search for in *source-expression*. The *substring-expression* cannot be NULL. See Character Value Expressions on page 6-41.

source-expression

is an SQL character value expression that specifies the source string. The *source-expression* cannot be NULL. See Character Value Expressions on page 6-41.

NonStop SQL/MX returns the result as a 2-byte signed integer with a scale of zero. If *substring-expression* is not found in *source-expression*, NonStop SQL/MX returns zero.

Considerations for POSITION

Result of POSITION

If the length of *source-expression* is zero and the length of *substring-expression* is greater than zero, NonStop SQL/MX returns 0. If the length of *substring-expression* is zero, NonStop SQL/MX returns 1.

If the length of *substring-expression* is greater than the length of *source-expression*, NonStop SQL/MX returns zero. If *source-expression* is a null value, NonStop SQL/MX returns a null value.

Using the UPSHIFT Function

To ignore case in the search, use the UPSHIFT function (or the LOWER function) for both the *substring-expression* and the *source-expression*.

Examples of POSITION

- This function returns the value 8 for the position of the substring `'John'` within the string:

  ```
  POSITION ('John' IN 'Robert John Smith')
  ```

- Suppose that the EMPLOYEE table has an EMPNAME column that contains both the first and last names. Return all records in table EMPLOYEE that contain the substring `'Smith'` regardless of whether the column value is in uppercase or lowercase characters:

  ```
  SELECT * FROM persnl.employee
    WHERE POSITION ('SMITH' IN UPSHIFT(empname)) > 0 ;
  ```

POWER Function

The POWER function returns the value of a numeric value expression raised to the power of an integer value expression. You can also use the exponential operator **.

POWER is an SQL/MX extension.

```
POWER (numeric-expression-1,numeric-expression-2)
```

numeric-expression-1,numeric-expression-2

are SQL numeric value expressions that specify the values for the base and exponent arguments of the POWER function. See Numeric Value Expressions on page 6-52.

If base *numeric-expression-1* is zero, the exponent *numeric-expression-2* must be greater than zero, and the result is zero. If the exponent is zero, the base cannot be 0, and the result is 1. If the base is negative, the exponent must be a value with an exact numeric data type and a scale of zero.

Examples of POWER

- Return the value 15.625:

  ```
  POWER (2.5,3)
  ```

- Return the value 27. The function POWER raised to the power of 2 is the inverse of the function SQRT:

  ```
  POWER (SQRT(27),2)
  ```

QUARTER Function

The QUARTER function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 4 that represents the corresponding quarter of the year. Quarter 1 represents January 1 through March 31, and so on.

QUARTER is an SQL/MX extension.

```
QUARTER (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of QUARTER

- Return an integer that represents the quarter of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, QUARTER(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
1996-04-10  1996-04-21 08:15:00.000000       2
```

QUERYCACHE Function

Considerations for QUERYCACHE
Examples of QUERYCACHE

The query plan cache automatically collects statistics regarding its use. When invoked, the QUERYCACHE table-valued stored function collects and returns the current state of these statistics in a single row table. The statistics are reinitialized when an `mxcmp` session is started and each `mxcmp` session maintains an independent set of statistics.

The QUERYCACHE function is an SQL/MX extension.

```
QUERYCACHE ()
```

The QUERYCACHE function can be specified as a table reference (*`table`*) in the FROM clause of a SELECT statement if it is preceded by the keyword TABLE and surrounded by parentheses. The syntax for the QUERYCACHE function has no parameters.

In a dynamic environment (that is, MXCI, MXCS, JDBC, or dynamic SQL), the QUERYCACHE function returns the statistics of the query plan cache of the `mxcmp` associated with the dynamic session. In a static environment (that is, statically compiled embedded SQL), the QUERYCACHE function returns zero rows because at runtime there is no associated `mxcmp`.

Considerations for QUERYCACHE

Using QUERYCACHE and DISPLAY_QC

The result of the QUERYCACHE function can be generated and displayed either by using the QUERYCACHE function or the DISPLAY_QC command. The DISPLAY_QC command provides a subset of the information displayed by the QUERYCACHE function. The output of the QUERYCACHE function and DISPLAY_QC command is machine-readable format. See the DISPLAY_QC Command on page 4-19.

Result of the QUERYCACHE Function

The result table of the QUERYCACHE function describes the query plan caching information for certain SELECT, INSERT, DELETE, UPDATE, or join statements. For more information about the types of statements that are appropriate candidates for query plan caching, see the *SQL/MX Query Guide*.

Column Name	**Data Type**	**Description**
CURRENT_SIZE	INT	Current size of the text and template cache.
MAX_CACHE_SIZE	INT	Maximum cache size in KB.
MAX_NUM_VICTIMS	INT	Maximum number of template cache entries that can be displaced from the cache to make room for a new entry. This number is the current QUERY_CACHE_MAX_VICTIMS CQD setting.
NUM_ENTRIES	INT	Number of template cache entries.
NUM_PINNED	INT	Total number of pinned entries.
NUM_COMPILES	INT	Number of complete SQL compile requests (excludes DESCRIBE and SHOWSHAPE requests).
NUM_RECOMPILES	INT	Number of recompilations. Recompilation refreshes a stale cache entry. Schema changes can cause cached queries to become stale.
NUM_RETRIES	INT	Number of successful compiles that succeed with caching off but fail with caching on. Report any occurrence to HP support.
NUM_CACHEABLE_ PARSING	INT	Number of compiled queries that `mxcmp` has processed after parsing and before binding of the query and that satisfy the conditions for caching.
NUM_CACHEABLE_ BINDING	INT	Number of compiled queries that `mxcmp` has processed after binding and before transformation of the query and that satisfy the conditions for caching.
NUM_CACHE_HITS_ PARSING	INT	Number of queries that `mxcmp` has compiled as cache hits after parsing.
NUM_CACHE_HITS_ BINDING	INT	Number of queries that `mxcmp` has compiled as cache hits after binding.
NUM_CACHEABLE_TOO_ LARGE	INT	Number of cacheable queries compiled by `mxcmp` that satisfy the conditions for caching but have plans that are too large to fit in the cache or have displaced too many cache entries (value > QUERY_CACHE_MAX_VICTIMS).
NUM_DISPLACED	INT	Number of entries displaced by template cache entries (to make room for new entries or displaced as a result of resizing of the cache).

Column Name	Data Type	Description
OPTIMIZATION_LEVEL	CHAR(10)	The current desired level of query optimization. Can be 0, 2, 3, or 5.
AVG_TEMPLATE_SIZE	INT UNSIGNED	Average template cache entry size in bytes. The size of a shared plan object is counted once.
TEXT_CACHE_HITS	INT UNSIGNED	Reserved for future use.
AVG_TEXT_SIZE	INT UNSIGNED	Reserved for future use.
TEXT_ENTRIES	INT UNSIGNED	Reserved for future use.
DISPLACED_TEXTS	INT UNSIGNED	Reserved for future use.
NUM_LOOKUPS	INT UNSIGNED	Reserved for future use.

Examples of QUERYCACHE

- Display all query plan caching statistics for an `mxcmp` session. Note that the output has been formatted for readability:

```
>>SET SCHEMA SAMDBCAT.PERSNL;

--- SQL operation complete.

>SELECT * FROM EMPLOYEE;

Employee/Number  First Name    Last Name          Dept/Num  Job/Code  Salary
---------------  ------------  -----------------  --------  --------  ------
              1  ROGER         GREEN               9000       100  175500.00
             23  JERRY         HOWARD              1000       100  137000.10
             29  JANE          RAYMOND             3000       100  136000.00
             32  THOMAS        RUDLOFF             2000       100  138000.40
.
.
.
--- 62 row(s) selected.

>SELECT * FROM TABLE (QUERYCACHE ());

AVG_PLAN_SIZE  31
CURRENT_SIZE  35
MAX_CACHE_SIZE  1024
MAX_NUM_VICTIMS  10
NUM_ENTRIES  1
NUM_PINNED  0
NUM_COMPILES  21
NUM_RECOMPILES  0
NUM_RETRIES  0
NUM_CACHEABLE_PARSING  0
NUM_CACHEABLE_BINDING  1
NUM_CACHE_HITS_PARSING  0
NUM_CACHE_HITS_BINDING  0
```

```
NUM_PIN_HITS_PARSING  0
NUM_PIN_HITS_BINDING  0

NUM_CACHEABLE_TOO_LARGE  0
NUM_DISPLACED  0
OPTIMIZATION_LEVEL  3
PINNING_STATEOFF

--- 1 row(s) selected.
```

- Display selected query plan caching statistics from the same query:

```
SELECT NUM_RETRIES, OPTIMIZATION_LEVEL FROM TABLE
    (QUERYCACHE ());

NUM_RETRIES  OPTIMIZATION_LEVEL
----------  ------------------
         0  3

--- 1 row(s) selected.
```

QUERYCACHEENTRIES Function

The query plan cache automatically collects statistics on each entry of the cache. When invoked, the QUERYCACHEENTRIES table-valued function collects and returns these statistics in a table with one row for each entry of the cache. The statistics are reinitialized when an `mxcmp` session is started. Each `mxcmp` session maintains an independent set of statistics.

The QUERYCACHEENTRIES function is an SQL/MX extension.

```
QUERYCACHEENTRIES ()
```

The QUERYCACHEENTRIES function can be specified as a table reference (*`table`*) in the FROM clause of a SELECT statement if it is preceded by the keyword TABLE and surrounded by parentheses. The syntax for the QUERYCACHEENTRIES function has no parameters.

In a dynamic environment (that is, MXCI, MXCS, JDBC, or dynamic SQL), the QUERYCACHEENTRIES function returns the statistics of each entry of the query plan cache of the `mxcmp` associated with the dynamic session. In a static environment (that is, statically compiled embedded SQL), the QUERYCACHEENTRIES function returns zero rows because at runtime there is no associated `mxcmp`.

Considerations for QUERYCACHEENTRIES

Using QUERYCACHEENTRIES and DISPLAY_QC_ENTRIES

The result of the QUERYCACHEENTRIES function can be generated and displayed either by using the QUERYCACHEENTRIES function or the DISPLAY_QC_ENTRIES command. The DISPLAY_QC_ENTRIES command displays a subset of the information contained in the QUERYCACHEENTRIES function. The output for the QUERYCACHEENTRIES function and DISPLAY_QC_ENTRIES is machine-readable format. See the DISPLAY_QC_ENTRIES Command on page 4-21.

Result of the QUERYCACHEENTRIES Function

The result table of the QUERYCACHEENTRIES function describes the query plan caching information for each entry in the query plan cache. For more information about the types of statements that are appropriate candidates for query plan caching, see the *SQL/MX Query Guide*.

Column Name	Data Type	Description
ROW_ID	INT	A zero-based sequential number. Entry number 0 is the most recently used entry. When a new entry is cached or matches the query issued, it occupies zero and all other cache entries not displaced are increased by one. Entry number 1 is the most recently used entry after the most recent (0). Entries with the highest row IDs are replaced; they are the least recently used entries.
PLAN_ID	LARGEINT	Primary Key. System-generated timestamp stored within each plan that uniquely identifies it. This column appears in the EXPLAIN table and enables joins between the two tables. Plan sharing can be recognized when the same PLAN_ID appears on multiple cache entries whose PHASE column is BINDING.
TEXT	CHAR(1024)	Text of the original SQL statement.
ENTRY_SIZE	INT	Size in bytes of this entry, excluding the size of the compiled plan with which this entry is associated.
NUM_HITS	INT	The total number of queries that have an identical query template with this entry and have reused the compiled plan.
PHASE	CHAR(10)	The `mxcmp` phase after when the plan associated with this entry was cached (parsing or binding). For template cache entries, the value is always binding.
OPTIMIZATION_LEVEL	CHAR(10)	The desired level of code optimization at the time the query was compiled. Can be 0, 2, 3, or 5.
CATALOG_NAME	CHAR(40)	Name of the default catalog under which the query was compiled.
SCHEMA_NAME	CHAR(40)	Name of the default schema under which the query was compiled.
NUM_PARAMS	INT	Number of constants in the query that were changed internally into parameters during compilation.
PARAM_TYPES	CHAR(1024)	Comma-separated list of the types of constants that were changed into parameters. Blank, if none.

Column Name	Data Type	Description
PLAN_LENGTH	INT	Size in bytes of the compiled plan associated with this entry.
COMPILATION_TIME	INT	Time in milliseconds it took to compile the query associated with this entry.
AVERAGE_HIT_TIME	INT	Time in milliseconds it took on the average to process a query as a cache hit against this entry.
SHAPE	CHAR(1024)	Required CONTROL QUERY SHAPE of the query associated with this entry. Blank if no required shape.
ISOLATION_LEVEL	CHAR(20)	Transaction isolation level associated with the query. Can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE, or NOT SPECIFIED.
ISOLATION_LEVEL_FOR_UPDATES	CHAR (20)	Transaction isolation level associated with the DELETE or UPDATE part of this query (if any) or with an INSERT statement. Can be READ_COMMITTED, REPEATABLE READ, or SERIALIZABLE.
ACCESS_MODE	CHAR(20)	Transaction access mode value associated with the query. Can be READ ONLY, READ WRITE, or NOT SPECIFIED.
AUTO_COMMIT	CHAR(15)	Transaction autocommit value associated with the query. Can be ON, OFF, or NOT SPECIFIED.
ROLLBACK_MODE	CHAR(15)	Transaction rollback mode value associated with the query. Can be WAITED, NO WAITED, or NOT SPECIFIED.

Examples of QUERYCACHEENTRIES

- Display all information contained in the QUERYCACHEENTRIES table. Note that the output has been formatted for readability.

```
>>SET SCHEMA SAMDBCAT.PERSNL;

--- SQL operation complete.

>SELECT * FROM EMPLOYEE;

Employee/Number First Name       Last Name      Dept/Num  Job/Code  Salary
--------------  --------------  --------------  --------  --------  ---------
             1  ROGER           GREEN             9000       100    175500.00
            23  JERRY           HOWARD            1000       100    137000.10
            29  JANE            RAYMOND           3000       100    136000.00
.
.
.
--- 62 row(s) selected.


>SELECT * FROM DEPT;
```

```
Dept/Num  Dept/Name     Mgr    Rpt/Dept  Location
--------  ------------  -----  --------  ------------------
    1000  FINANCE          23      9000  CHICAGO
    1500  PERSONNEL       213      1000  CHICAGO
    2000  INVENTORY        32      9000  LOS ANGELES
.
.
.
--- 12 row(s) selected.


>SELECT * FROM JOB;

Job/Code  Job Description
--------  ------------------

     100  MANAGER
     200  PRODUCTION SUPV
     250  ASSEMBLER
.
.
.
--- 10 row(s) selected.

>SELECT * FROM TABLE (QUERYCACHEENTRIES ());

ROW_ID    PLAN_ID             TEXT                            ENTRY_SIZE
------    ------------------  --------------------------      ----------
     0    211894097543468116  select * from employee;              32410
     1    211894097552547493  select * from job;                   24968
     2    211894097548497817  select * from dept;                  29730

NUM_HITS    PHASE        OPTIMIZATION_LEVEL       CATALOG_NAME   SCHEMA_NAME
--------    ----------   ----------------------     ------------   -----------
       1     BINDING      3                         SAMDBCAT       PERSNL
       0     BINDING      3                         SAMDBCAT       PERSNL
       0     BINDING      3                         SAMDBCAT       PERSNL

NUM_PARAMS    PARAM_TYPES    PLAN_LENGTH  IS_PINNED  COMPILATION_TIME
----------    -----------    -----------  ---------  -----------------
       0                           31752     OFF                334
       0                           24504     OFF                54
       0                           29144     OFF                96

AVERAGE_HIT_TIME    SHAPE    ISOLATION_LEVEL     ACCESS_MODE     AUTO_COMMIT
----------------    -------  ---------------     ------------    -----------
41                            READ COMMITTED     READ/WRITE         ON
0                             READ COMMITTED     READ/WRITE         ON
0                             READ COMMITTED     READ/WRITE         ON

ROLLBACK_MODE
--------------
NOT SPECIFIED
NOT SPECIFIED
NOT SPECIFIED
```

- Display selected columns of the QUERYCACHEENTRIES table using the same cached queries:

```
>>SELECT PLAN_ID, NUM_HITS, IS_PINNED FROM TABLE
(QUERYCACHEENTRIES ());

PLAN_ID                    NUM_HITS    IS_PINNED
```

```
-------------------  ----------   ---------
211894097543468116            1    OFF
211894097552547493            0    OFF
211894097548497817            0    OFF

--- 3 row(s) selected.
```

RADIANS Function

The RADIANS function converts a numeric value expression expressed in degrees to the number of radians.

RADIANS is an SQL/MX extension.

```
RADIANS (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the RADIANS function. See Numeric Value Expressions on page 6-52.

Examples of RADIANS

- Return the value 7.85398150000000160E-001, or approximately 0.78540 in degrees:

  ```
  RADIANS (45)
  ```

- Return the value 45 in degrees. The function DEGREES is the inverse of the function RADIANS.

  ```
  DEGREES (RADIANS (45))
  ```

RELATEDNESS Function

RELATEDNESS is a built-in table-valued function that returns relatedness information for a single entity.

```
relatedness ('E_TYPE', 'E_VALUE')
```

Table 8-3 shows the input and output parameters for RELATEDNESS.

Table 8-3. Input and Output Parameters for RELATEDNESS

Input/Output Type	Parameter	Specification	Description
Input parameter	E_TYPE	CHAR (32) NOT NULL	The type of version information that is requested.
Input parameter	E_VALUE	VARCHAR(517) NOT NULL	The name of the entity for which version information is requested. The type of that entity is implied by E_TYPE.
Output column	E_TYPE	CHAR (32) NOT NULL	A copy of the actual value for the E_TYPE input parameter.
Output column	E_VALUE	VARCHAR(517) NOT NULL	A copy of the actual value for the E_VALUE input parameter.
Output column	NAME	VARCHAR(517) NOT NULL	The name of a related entity.

ANSI names in the input value parameter must be fully qualified in external format. Expand node names are case-insensitive. Both input parameters must be character-valued expressions.

Example of RELATEDNESS

```
select * from table (relatedness ('SCHEMA', 'CAT.SCH'));

E_TYPE      E_VALUE         NAME
-------     ---------       ----------------------
SCHEMA      CAT.SCH         CAT.SCH
SCHEMA      CAT.SCH         OTHERCAT.OTHERSCH
SCHEMA      CAT.SCH         YET_ANOTHER_CAT.SCHEMAX
```

REPEAT Function

The REPEAT function returns a character string composed of the evaluation of a character expression repeated a specified number of times.

REPEAT is an SQL/MX extension.

```
REPEAT (character-expr, count)
```

character-expr

specifies the source string from which to return the specified number of repeated strings. The source string is an SQL character value expression. The operand is the result of evaluating *character-expr*. See Character Value Expressions on page 6-41.

count

specifies the number of times the source string *character-expr* is to be repeated. The number *count* must be a value greater than or equal to zero of exact numeric data type and with a scale of zero.

Examples of REPEAT

- Return this quote from Act 5, Scene 3, of King Lear:

```
REPEAT ('Never,', 5)
```

```
Never,Never,Never,Never,Never,
```

REPLACE Function

The REPLACE function returns a character string where all occurrences of a specified character string in the original string are replaced with another character string.

REPLACE is an SQL/MX extension.

```
REPLACE (char-expr-1, char-expr-2, char-expr-3)
```

char-expr-1, char-expr-2, char-expr-3

are SQL character value expressions. The operands are the result of evaluating the character expressions. All occurrences of *char-expr-2* in *char-expr-1* are replaced by *char-expr-3*. See Character Value Expressions on page 6-41.

Examples of REPLACE

- Use the REPLACE function to change job descriptions so that occurrences of the company name are updated:

```
SELECT jobdesc FROM persnl.job;

Job Description
------------------

MANAGER COMNET
PRODUCTION  COMNET
ASSEMBLER COMNET
SALESREP COMNET
SYSTEM ANAL COMNET
...

--- 10 row(s) selected.

UPDATE persnl.job
SET jobdesc = REPLACE (jobdesc, 'COMNET', 'TDMNET');

Job Description
------------------

MANAGER TDMNET
PRODUCTION  TDMNET
ASSEMBLER TDMNET
SALESREP TDMNET
SYSTEM ANAL TDMNET
...

--- 10 row(s) selected.
```

RIGHT Function

The RIGHT function returns the rightmost specified number of characters from a character expression.

RIGHT is an SQL/MX extension.

```
RIGHT (character-expr, count)
```

character-expr

specifies the source string from which to return the rightmost specified number of characters. The source string is an SQL character value expression. The operand is the result of evaluating *character-expr*. See Character Value Expressions on page 6-41.

count

specifies the number of characters to return from *character-expr*. The number *count* must be a value of exact numeric data type with a scale of zero.

Examples of RIGHT

- Return 'Smith':

  ```
  RIGHT ('Robert John Smith', 5)
  ```

- Suppose that a six-character company literal has been concatenated as the first six characters to the job descriptions in the JOB table. Use the RIGHT function to remove the company literal from the job descriptions:

  ```
  UPDATE persnl.job
  SET jobdesc = RIGHT (jobdesc, 12);
  ```

ROWS SINCE Function

The ROWS SINCE function is a sequence function that returns the number of rows counted since the specified condition was last true in the intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

ROWS SINCE is an SQL/MX extension.

```
ROWS SINCE [INCLUSIVE] (condition [,max-rows])
```

`INCLUSIVE`

specifies the current row is to be considered. If you specify INCLUSIVE, the condition is evaluated in the current row. Otherwise, the condition is evaluated beginning with the previous row. If you specify INCLUSIVE and the condition is true in the current row, ROWS SINCE returns 0.

`condition`

specifies a condition to be considered for each row in the result table. Each column in *`condition`* must be a column that exists in the result table. If the condition has never been true for the result table, ROWS SINCE returns null.

`max-rows`

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows from the current row to consider. If the condition has never been true for *`max-rows`* from the current row, or if *`max-rows`* is negative or null, ROWS SINCE returns null.

Considerations for ROWS SINCE

Counting the Rows

If you specify INCLUSIVE, the condition in each row of the result table is evaluated starting with the current row as row 0 (zero) (up to the maximum number of rows or the size of the result table). Otherwise, the condition is evaluated starting with the previous row as row 1.

If a row is reached where the condition is true, ROWS SINCE returns the number of rows counted so far. Otherwise, if the condition is never true within the result table being considered, ROWS SINCE returns null. NonStop SQL/MX then goes to the next row as the new current row.

Examples of ROWS SINCE

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,I2 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1, I2, and TS with data that is sequenced by column TS:

I1	I2	TS
6215	7516	TIMESTAMP '1950-03-05 08:32:09'
null	497	TIMESTAMP '1951-02-15 14:35:49'
19058	26165	TIMESTAMP '1955-05-18 08:40:10'
null	9681	TIMESTAMP '1960-09-19 14:40:39'
11966	12356	TIMESTAMP '1964-05-01 16:41:02'

- Return the number of rows since the condition I1 IS NULL became true:

```
SELECT ROWS SINCE (I1 IS NULL) AS ROWS_SINCE_NULL
FROM mining.seqfcn
SEQUENCE BY TS;

ROWS_SINCE_NULL
---------------
              ?
              ?
              1
              2
              1

--- 5 row(s) selected.
```

- Return the number of rows since the condition I1 < I2 became true:

```
SELECT ROWS SINCE (I1<I2), ROWS SINCE INCLUSIVE (I1<I2)
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)           (EXPR)
---------------  ---------------
              ?                0
              1                1
              2                0
              1                1
              2                0

--- 5 row(s) selected.
```

RPAD Function

The RPAD function replaces the rightmost specified number of characters in a character expression with a padding character or string. With the ANSI_STRING_FUNCTIONALITY CQD set to ON, the function pads the right side of a character expression with the specified string.

```
RPAD (character-expr,count [,pad-character])
```

character-expr

is an SQL character value expression. The operand is the result of evaluating *character-expr*. See Character Value Expressions on page 6-41.

count

specifies the number of characters. The *count* must be greater than or equal to zero of exact numeric data type and with a scale of zero. For considerations of *count* based on CQD ANSI_STRING_FUNCTIONALITY, see Examples of RPAD on page 8-150.

pad-character

specifies the padding character or a string. If no *pad-character* is specified, spaces is the padding character. For KANJI or KSC5601, the code value of *pad-character* is hexadecimal 2020.

Examples of RPAD

The behavior of the RPAD function when the ANSI_STRING_FUNCTIONALITY CQD is set to ON and the corresponding examples are described below.

The *count* specifies the number of characters to be returned. It is the length of the result string.

If *count* is smaller than the length of the *character-expr*, the *character-expr* is truncated. If *count* is equal to the length of the *character-expr*, the value of the *character-expr* is retained. If *count* is greater than the length of the *character-expr*, the *character-expr* is right-padded with the pad-character.

- The following RPAD function truncates the string '`kite`' because the specified *count* is less than the string size and returns '`ki`':

  ```
  RPAD('kite', 2)
  ```

- The following RPAD function returns the original string '`go fly a kite`' because *count* is equal to length of the string:

  ```
  RPAD('go fly a kite', 13, 'z')
  ```

- The following RPAD function returns a string of seven characters '`kite   `'. The string '`kite`' is right-padded with three spaces.

```
RPAD('kite', 7)
```

- The following RPAD function returns a string of eight characters '`kite0000`'. The string '`kite`' is right-padded with four pad-characters '`0`'.

```
RPAD('kite', 8, '0')
```

- The following RPAD function returns a string of 14 characters '`go fly a kitez`'. The string '`go fly a kite`' is right-padded with one pad-character '`z`'.

```
RPAD('go fly a kite', 14, 'z')
```

- The following RPAD function returns a string of 17 characters '`kitegoflygoflygof`'. The string '`kite`' is right-padded with string '`gofly`' such that the result string has 17 characters.

```
RPAD('kite', 17, 'gofly' )
```

The default behavior of the RPAD function and the corresponding examples are described below.

The *`count`* specifies the number of characters to be replaced. The *`count`* must be less than or equal to the length of the *`character-expr`*. If *`count`* is smaller than or equal to the length of the *`character-expr`*, the rightmost *`count`* characters of the *`character-expr`* are replaced with the padding characters or string. If *`count`* is greater than the length of the *`character-expr`*, an error is returned.

- The following RPAD function replaces two rightmost characters in the string '`kite`' with spaces and returns '`ki  `':

```
RPAD('kite', 2)
```

- The following RPAD function replaces two rightmost characters '`Jo`' with the string '`John,`' twice and returns '`go fly a kite John,John,`':

```
RPAD('go fly a kite Jo', 2, 'John,')
```

- The following RPAD function replaces 13 rightmost characters in the string '`go fly a kite`' with character '`z`' and returns '`zzzzzzzzzzzzz`':

```
RPAD('go fly a kite', 13, 'z')
```

- The following RPAD functions return an error because the *`count`* is greater than the string length:

```
RPAD('kite', 7)
RPAD('kite', 8, '0')
RPAD('go fly a kite', 14, 'z')
RPAD('kite', 17, 'gofly' )
```

RTRIM Function

The RTRIM function removes the specified trim characters from the right side of the character string. If the trim characters are not specified, by default, the function removes spaces.

```
RTRIM(srcstr[, trim_chrs])
```

srcstr

is a SQL character value expression from which spaces or *trim_chrs* are removed from the right side. See Character Value Expressions on page 6-41.

trim_chrs

is the character or characters to be removed from the *srcstr* . The default is the space character.

Considerations for RTRIM

Result of RTRIM

The result is always of type NON ANSI VARCHAR, with maximum length equal to the fixed length or maximum variable length of *srcstr*.

Examples of RTRIM

- Return `'    Robert'`:

  `RTRIM ('    Robert    ')`

  See TRIM Function on page 8-192 and LTRIM Function on page 8-102.

- In this example, characters '1' and '0' are removed from the right side of the `colA` value:

```
create table tab1(colA varchar(20), colB int);
insert into tab1 values('ten0101010101', 10);
insert into tab1 values('nine11001101', 9);
select rtrim(colA,'10'), colB from tab1;
   (EXPR)                     COLB
--------------------          -----------
ten                                    10
nine                                    9

--- 2 row(s) selected
```

RUNNINGAVG Function

The RUNNINGAVG function is a sequence function that returns the average of non-null values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

RUNNINGAVG is an SQL/MX extension.

```
RUNNINGAVG (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGAVG returns the average of non-null values of *column-expression* up to and including the current row.

Considerations for RUNNINGAVG

Equivalent Result

The result of RUNNINGAVG is equivalent to:

```
RUNNINGSUM(column-expr) / RUNNINGCOUNT(*)
```

Examples of RUNNINGAVG

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the average of non-null values of I1 up to and including the current row:

```
SELECT RUNNINGAVG (I1) AS AVG_I1
FROM mining.seqfcn
SEQUENCE BY TS;

AVG_I1
--------------------
                6215
               17194
               11463
                9746
               10190

--- 5 row(s) selected.
```

RUNNINGCOUNT Function

The RUNNINGCOUNT function is a sequence function that returns the number of rows up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

RUNNINGCOUNT is an SQL/MX extension.

```
RUNNINGCOUNT {(*) | (column-expression)}
```

`*`

as an argument causes RUNNINGCOUNT(*) to return the number of rows in the intermediate result table up to and including the current row.

`column-expression`

specifies a derived column determined by the evaluation of the column expression. If *`column-expression`* is the argument, RUNNINGCOUNT returns the number of rows containing non-null values of *`column-expression`* in the intermediate result table up to and including the current row.

Considerations for RUNNINGCOUNT

No DISTINCT Clause

The RUNNINGCOUNT sequence function is defined differently from the COUNT aggregate function. If you specify DISTINCT for the COUNT aggregate function, duplicate values are eliminated before COUNT is applied. Note that you cannot specify DISTINCT for the RUNNINGCOUNT sequence function; duplicate values are counted.

Examples of RUNNINGCOUNT

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	**TS**
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the number of rows up to and including the current row:

```
SELECT RUNNINGCOUNT (*) AS COUNT_ROWS
FROM mining.seqfcn
SEQUENCE BY TS;

COUNT_ROWS
------------
           1
           2
           3
           4
           5

--- 5 row(s) selected.
```

- Return the number of rows that include non-null values of I1 up to and including the current row:

```
SELECT RUNNINGCOUNT (I1) AS COUNT_I1
FROM mining.seqfcn
SEQUENCE BY TS;

COUNT_I1
------------
           1
           2
           2
           3
           4

--- 5 row(s) selected.
```

RUNNINGMAX Function

The RUNNINGMAX function is a sequence function that returns the maximum of values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

RUNNINGMAX is an SQL/MX extension.

```
RUNNINGMAX (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGMAX returns the maximum of values of *column-expression* up to and including the current row.

Examples of RUNNINGMAX

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
19058	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the maximum of values of I1 up to and including the current row:

```
SELECT RUNNINGMAX (I1) AS MAX_I1
FROM mining.seqfcn
SEQUENCE BY TS;

MAX_I1
------------
        6215
       28174
       28174
       28174
       28174

--- 5 row(s) selected.
```

RUNNINGMIN Function

The RUNNINGMIN function is a sequence function that returns the minimum of values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

RUNNINGMIN is an SQL/MX extension.

```
RUNNINGMIN (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGMIN returns the minimum of values of *column-expression* up to and including the current row.

Examples of RUNNINGMIN

Suppose that SEQFCN has been created as:

```
CREATE $db.table mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
19058	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the minimum of values of I1 up to and including the current row:

```
SELECT RUNNINGMIN (I1) AS MIN_I1
FROM mining.seqfcn
SEQUENCE BY TS;

MIN_I1
------------
        6215
        6215
        6215
        4597
        4597

--- 5 row(s) selected.
```

RUNNINGSTDDEV Function

The RUNNINGSTDDEV function is a sequence function that returns the standard deviation of non-null values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

RUNNINGSTDDEV is an SQL/MX extension.

```
RUNNINGSTDDEV (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGSTDDEV returns the standard deviation of non-null values of *column-expression* up to and including the current row.

Considerations for RUNNINGSTDDEV

Equivalent Result

The result of RUNNINGSTDDEV is equivalent to:

```
SQRT(RUNNINGVARIANCE(column-expression))
```

Examples of RUNNINGSTDDEV

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the standard deviation of non-null values of I1 up to and including the current row:

```
SELECT RUNNINGSTDDEV (I1) AS STDDEV_I1
FROM mining.seqfcn
SEQUENCE BY TS;

STDDEV_I1
-------------------------
 0.00000000000000000E+000
 1.55273578080753976E+004
 1.48020166531456112E+004
 1.25639147428923072E+004
 1.09258501408357232E+004

--- 5 row(s) selected.
```

Note that you can use the CAST function for display purposes. For example:

```
SELECT CAST(RUNNINGSTDDEV (I1) AS DEC (18,3))
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)
--------------------
                .000
           15527.357
           14802.016
           12563.914
           10925.850

--- 5 row(s) selected.
```

RUNNINGSUM Function

The RUNNINGSUM function is a sequence function that returns the sum of non-null values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

RUNNINGSUM is an SQL/MX extension.

```
RUNNINGSUM (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGSUM returns the sum of non-null values of *column-expression* up to and including the current row.

Examples of RUNNINGSUM

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the sum of non-null values of I1 up to and including the current row:

```
SELECT RUNNINGSUM (I1) AS SUM_I1
FROM mining.seqfcn
SEQUENCE BY TS;

SUM_I1
--------------------
                6215
               34389
               34389
               38986
               50952

--- 5 row(s) selected.
```

RUNNINGVARIANCE Function

The RUNNINGVARIANCE function is a sequence function that returns the variance of non-null values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See SEQUENCE BY Clause on page 7-19.

RUNNINGVARIANCE is an SQL/MX extension.

```
RUNNINGVARIANCE (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGVARIANCE returns the variance of non-null values of *column-expression* up to and including the current row.

Examples of RUNNINGVARIANCE

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
 (I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
null	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
11966	TIMESTAMP '1964-05-01 16:41:02'

- Return the variance of non-null values of I1 up to and including the current row:

```
SELECT RUNNINGVARIANCE (I1) AS VARIANCE_I1
FROM mining.seqfcn
SEQUENCE BY TS;

VARIANCE_I1
-------------------------
 0.00000000000000000E+000
 2.41098840499999960E+008
 2.19099696999999968E+008
 1.57851953666666640E+008
 1.19374201299999980E+008

--- 5 row(s) selected.
```

Note that you can use the CAST function for display purposes. For example:

```
SELECT CAST(RUNNINGVARIANCE (I1) AS DEC (18,3))
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)
--------------------
                .000
       241098840.500
       219099697.000
       157851953.666
       119374201.299

--- 5 row(s) selected.
```

SECOND Function

The SECOND function converts a TIME or TIMESTAMP expression into an INTEGER value in the range 0 through 59 that represents the corresponding second of the hour.

SECOND is an SQL/MX extension.

```
SECOND (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type TIME or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of SECOND

- Return an integer that represents the second of the hour from the SHIP_TIMESTAMP column:

```
SELECT start_date, ship_timestamp, SECOND(ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  -----------
1996-04-10  1996-04-21 08:15:00.000000      .000000
```

SESSION_USER Function

The SESSION_USER function returns the current Guardian user ID as variable-length character data in the form *`group.name`*.

```
SESSION_USER
```

The SESSION_USER function is equivalent to the CURRENT_USER Function on page 8-43 and the USER Function on page 8-203

Examples of SESSION_USER

- Use SESSION_USER to display the current session user:

```
SELECT SESSION_USER FROM logfile;

(EXPR)
-----------------------
DCS.TSHAW
...

--- 5 row(s) selected.
```

SIGN Function

The SIGN function returns an indicator of the sign of a numeric value expression. If the value is less than zero, the function returns -1 as the indicator. If the value is zero, the function returns 0. If the value is greater than zero, the function returns 1.

SIGN is an SQL/MX extension.

```
SIGN (numeric-expression)
```

`numeric-expression`

is an SQL numeric value expression that specifies the value for the argument of the SIGN function. See Numeric Value Expressions on page 6-52.

Examples of SIGN

- Return the value –1:

```
SIGN (-20 + 12)
```

- Return the value 0:

```
SIGN (-20 + 20)
```

- Return the value 1:

```
SIGN (-20 + 22)
```

SIN Function

The SIN function returns the sine of a numeric value expression, where the expression is an angle expressed in radians.

SIN is an SQL/MX extension.

```
SIN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the SIN function. See Numeric Value Expressions on page 6-52.

Examples of SIN

- This function returns the value 3.42052233254419920E-001, or approximately 0.3420, the sine of 0.3491 (which is 20 degrees):

  ```
  SIN (0.3491)
  ```

SINH Function

The SINH function returns the hyperbolic sine of a numeric value expression, where the expression is an angle expressed in radians.

SINH is an SQL/MX extension.

```
SINH (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the SINH function. See Numeric Value Expressions on page 6-52.

Examples of SINH

- This function returns the value 1.60191908030082600E+000, or approximately 1.6019, the hyperbolic sine of 1.25:

  ```
  SINH (1.25)
  ```

SPACE Function

The SPACE function returns a character string consisting of a specified number of spaces each of which is 0x20 (for the ISO88591 character set), 0x0020 (for the UCS2 character set), or 0x2020 (for the KANJI and KSC5601 character sets).

SPACE is an SQL/MX extension.

```
SPACE (length [,char-set-name])
```

length

specifies the number of characters to be returned. The number *count* must be a value greater than or equal to zero of exact numeric data type and with a scale of zero. *length* cannot exceed 32768 for the ISO8859-1 character set or 16384 for the UCS2, KANJI, and KSC5601 character sets.

char-set-name

can be ISO88591, KANJI, KSC5601, or UCS2. The default is ISO88591.

The returned character string will be of data type VARCHAR associated with the character set specified by *char-set-name*.

Examples of SPACE

- Return 3 spaces:

  ```
  SPACE (3)
  ```

SQRT Function

The SQRT function returns the square root of a numeric value expression.

SQRT is an SQL/MX extension.

```
SQRT (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the SQRT function. The value of the argument must not be a negative number. See Numeric Value Expressions on page 6-52.

Examples of SQRT

- This function returns the value 5.19615242270663312E+000, or approximately 5.196:

  ```
  SQRT (27)
  ```

STDDEV Function

Considerations for STDDEV
Examples of STDDEV

STDDEV is an aggregate function that returns the standard deviation of a set of numbers.

STDDEV is an SQL/MX extension.

```
STDDEV ([ALL | DISTINCT] expression [,weight])
```

`ALL | DISTINCT`

specifies whether duplicate values are included in the computation of the STDDEV of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the STDDEV function is applied. If DISTINCT is specified, you cannot specify *weight*.

expression

specifies a numeric value expression that determines the values for which to compute the standard deviation. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the STDDEV function operates on distinct values from the one-column table derived from the evaluation of *expression*.

weight

specifies a numeric value expression that determines the weights of the values for which to compute the standard deviation. *weight* cannot contain an aggregate function or a subquery. *weight* is defined on the same table as *expression*. The one-column table derived from the evaluation of *expression* and the one-column table derived from the evaluation of *weight* must have the same cardinality.

Considerations for STDDEV

Definition of STDDEV

The standard deviation of a value expression is defined to be the square root of the variance of the expression. See VARIANCE Function on page 8-207.

Because the definition of variance has *N-1* in the denominator of the expression (if weight is not specified), NonStop SQL/MX returns a system-defined default setting of zero (and no error) if the number of rows in the table, or a group of the table, is equal to 1.

Data Type of the Result

The data type of the result is always DOUBLE PRECISION.

Operands of the Expression

The expression includes columns from the rows of the SELECT result table but cannot include an aggregate function. These are valid:

```
STDDEV (SALARY)
STDDEV (SALARY * 1.1)
STDDEV (PARTCOST * QTY_ORDERED)
```

Nulls

STDDEV is evaluated after eliminating all nulls from the set. If the result table is empty, STDDEV returns NULL.

FLOAT(54) and DOUBLE PRECISION Data

Avoid using large FLOAT(54) or DOUBLE PRECISION values as arguments to STDDEV. If SUM(x * x) exceeds the value of 1.15792089237316192e77 during the computation of STDDEV(x), a numeric overflow occurs.

Examples of STDDEV

- Compute the standard deviation of the salary of the current employees:

  ```
  SELECT STDDEV(salary) AS StdDev_Salary
  FROM persnl.employee;

  STDDEV_SALARY
  -------------------------
   3.57174062500000000E+004

  --- 1 row(s) selected.
  ```

- Compute the standard deviation of the cost of parts in the current inventory:

  ```
  SELECT STDDEV (price * qty_available)
  FROM sales.parts;

  (EXPR)
  -------------------------
   7.13899499999999808E+006

  --- 1 row(s) selected.
  ```

- Suppose that your database includes a WEATHER table, which is created by using SQLCI in this way:

```
CREATE TABLE $db.mining.weather
 ( city         VARCHAR (20) NO DEFAULT NOT NULL
  ,state        CHAR (2)     NO DEFAULT NOT NULL
  ,date_weather DATE NO DEFAULT NOT NULL
  ,temperature  NUMERIC (3) SIGNED
  ,weight       NUMERIC (2) UNSIGNED
  ,PRIMARY KEY (city, state, date_weather) )
  CATALOG $db.mining
  ORGANIZATION KEY SEQUENCED;
```

After the table is created, you can insert the mapping into the OBJECTS table by using MXCI in this way:

```
CREATE SQLMP ALIAS db.mining.weather $db.mining.weather;
```

Suppose that the WEATHER table contains these rows:

CITY	STATE	DATE_WEATHER	TEMPERATURE	WEIGHT
Austin	TX	1997-01-01	50	1
Austin	TX	1997-01-02	40	1
Austin	TX	1997-01-03	60	2
Austin	TX	1997-01-04	84	2
Cupertino	CA	1997-01-01	65	1
Cupertino	CA	1997-01-02	65	2
Cupertino	CA	1997-01-03	65	2
Cupertino	CA	1997-01-04	65	1

  - Find the standard deviation of the TEMPERATURE column by STATE:

```
SELECT state, STDDEV (temperature)
 FROM weather GROUP BY state;

STATE      (EXPR)
----------  --------------------
TX      1.88591308593750024E+001
CA      0.00000000000000000E+000

--- 2 row(s) selected.
```

SUBSTRING Function

The SUBSTRING function extracts a substring out of a given character expression. It returns a character string of data type VARCHAR, with maximum length equal to the fixed length or maximum variable length of the character expression.

```
SUBSTRING (character-expr FROM start-position [FOR length])

or:

SUBSTRING (character-expr,start-position,length)
```

character-expr

specifies the source string from which to extract the substring. The source string is an SQL character value expression. The operand is the result of evaluating *character-expr*. See Character Value Expressions on page 6-41.

start-position

specifies the starting position *start-position* within *character-expr* at which to start extracting the substring. *start-position* must be a value with an exact numeric data type and a scale of zero.

length

specifies the number of characters to extract from *character-expr*. *length* is the length of the extracted substring and must be a value greater than or equal to zero of exact numeric data type and with a scale of zero.

If you are using the FROM keyword, the *length* field is optional, therefore, if you do not specify the substring *length*, all characters starting at *start-position* and continuing until the end of the character expression are returned. If you are not using the FROM and FOR keywords, the *length* field is required.

Considerations for SUBSTRING

Requirements for the Expression, Length, and Start Position

- The data types of the substring length and the start position must be numeric with a scale of zero. Otherwise, an error is returned.
- If the sum of the start position and the substring length is greater than the length of the character expression, the substring from the start position to the end of the string is returned.
- If the start position is greater than the length of the character expression, an empty string ('') is returned.

- The resulting substring is always of type VARCHAR. If the source character string is an upshifted CHAR or VARCHAR string, the result is an upshifted VARCHAR type.

Examples of SUBSTRING

- Extract 'Ro':

```
SUBSTRING('Robert John Smith' FROM 0 FOR 3)
```

- Extract `'John'`:

```
SUBSTRING ('Robert John Smith' FROM 8 FOR 4)
```

- Extract `'John Smith'`:

```
SUBSTRING ('Robert John Smith' FROM 8)
```

- Extract `'Robert John Smith'`:

```
SUBSTRING ('Robert John Smith' FROM 1 FOR 17)
```

- Extract `'John Smith'`:

```
SUBSTRING ('Robert John Smith' FROM 8 FOR 15)
```

- Extract `'Ro'`:

```
SUBSTRING ('Robert John Smith' FROM -2 FOR 5)
```

- Extract an empty string `''`:

```
SUBSTRING ('Robert John Smith' FROM 8 FOR 0)
```

SUM Function

SUM is an aggregate function that returns the sum of a set of numbers.

```
SUM ([ALL | DISTINCT] expression)
```

`ALL | DISTINCT`

specifies whether duplicate values are included in the computation of the SUM of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the SUM function is applied.

expression

specifies a numeric or interval value expression that determines the values to sum. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the SUM function operates on distinct values from the one-column table derived from the evaluation of *expression*. All nulls are eliminated before the function is applied to the set of values. If the result table is empty, SUM returns NULL.

See Expressions on page 6-41.

Considerations for SUM

Data Type and Scale of the Result

The data type of the result depends on the data type of the argument. If the argument is an exact numeric type, the result is LARGEINT. If the argument is an approximate numeric type, the result is DOUBLE PRECISION. If the argument is INTERVAL data type, the result is INTERVAL with the same precision as the argument. The scale of the result is the same as the scale of the argument. If the argument has no scale, the result is truncated.

Operands of the Expression

The expression includes columns from the rows of the SELECT result table—but cannot include an aggregate function. The valid expressions are:

```
SUM (SALARY)
SUM (SALARY * 1.1)
SUM (PARTCOST * QTY_ORDERED)
```

Examples of SUM

- Compute the total value of parts in the current inventory:

```
SELECT SUM (price * qty_available)
FROM sales.parts;

(EXPR)
---------------------
         117683505.96

--- 1 row(s) selected.
```

TAN Function

The TAN function returns the tangent of a numeric value expression, where the expression is an angle expressed in radians.

TAN is an SQL/MX extension.

```
TAN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the TAN function. See Numeric Value Expressions on page 6-52.

Examples of TAN

- This function returns the value 3.64008908293626896E-001, or approximately 0.3640, the tangent of 0.3491 (which is 20 degrees):

  ```
  TAN (0.3491)
  ```

TANH Function

The TANH function returns the hyperbolic tangent of a numeric value expression, where the expression is an angle expressed in radians.

TANH is an SQL/MX extension.

```
TANH (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the TANH function. See Numeric Value Expressions on page 6-52.

Examples of TANH

- This function returns the value 8.48283639957513088E-001 or approximately 0.8483, the hyperbolic tangent of 1.25:

  ```
  TANH (1.25)
  ```

THIS Function

The THIS function is a sequence function that is used in the ROWS SINCE function to distinguish between the value of the column in the current row and the value of the column in previous rows (in an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement). See ROWS SINCE Function on page 8-148.

THIS is an SQL/MX extension.

```
THIS (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. If the value of the expression is null, THIS returns null.

Considerations for THIS

Counting the Rows

You can use the THIS function only within the ROWS SINCE function. For each row, the ROWS SINCE condition is evaluated in two steps:

1. The expression for THIS is evaluated for the current row. This value becomes a constant.
2. The condition is evaluated for the result table, using a combination of the THIS constant and the data for each row in the result table, starting with the previous row as row 1 (up to the maximum number of rows or the size of the result table).

If a row is reached where the condition is true, ROWS SINCE returns the number of rows counted so far. Otherwise, if the condition is never true within the result table being considered, ROWS SINCE returns null. NonStop SQL/MX then goes to the next row as the new current row and the THIS constant is reevaluated.

Example of THIS

Suppose that SEQFCN has been created as:

```
CREATE TABLE $db.mining.seqfcn
(I1 INTEGER,TS TIMESTAMP);
```

Within MXCI, the ANSI alias name has been mapped as:

```
CREATE SQLMP ALIAS db.mining.seqfcn $db.mining.seqfcn;
```

The table SEQFCN has columns I1 and TS with data that is sequenced by column TS:

I1	TS
6215	TIMESTAMP '1950-03-05 08:32:09'
28174	TIMESTAMP '1951-02-15 14:35:49'
19058	TIMESTAMP '1955-05-18 08:40:10'
4597	TIMESTAMP '1960-09-19 14:40:39'
31966	TIMESTAMP '1964-05-01 16:41:02'

- Return the number of rows since the condition I1 less than a previous row became true:

```
SELECT ROWS SINCE (THIS(I1) < I1) AS ROWS_SINCE_THIS
FROM mining.seqfcn
SEQUENCE BY TS;

ROWS_SINCE_THIS
---------------
              ?
              ?
              1
              1
              ?

--- 5 row(s) selected.
```

TO_CHAR(<NUMERIC>) Function

The TO_CHAR(`<numeric>`)function converts numeric data to formatted string. The result string is VARCHAR type.

The following is the function syntax:

```
TO_CHAR(numeric-expression, format-value)
```

numeric-expression

is an SQL numeric value expression. Numeric data type can be `NUMERIC(P,S)`, `SMALLINT`, `INT[EGER]`, `LARGEINT`, `DEC[IMAL]`, `REAL`, `FLOAT`, or `DOUBLE PRECISION`. NULL or dynamic parameter cannot be used as *numeric-expressions*.

format-value

is a constant string that defines the output format. NULL, column name, expression, or dynamic parameter cannot be defined as *format-value*. The following table lists all the numeric elements that can be used to construct the numeric *format-value*.

Element	**Description**	*numeric-expression*	*format-value*	**Output**
9	Specifies the number of digits in the output format.	1234	9999	‘ 1234’
0	Returns the value with leading zeroes.	1	0999	‘ 0001’
.(period) D	Specifies the position of the decimal point in the output format. Restriction: A decimal point can occur only once within a format-value.	12.34	99.99 99D99	‘ 12.34’ ‘ 12.34’
,(comma) G	Returns ‘,’ in the specified position. Restriction: You cannot specify this element at the beginning or after the decimal point.	1234	9,999 9G99	‘ 1,234’ ‘ 1,234’
$	Returns a leading dollar sign. Restriction: If you specify the dollar sign at any position in the *format-value*, the output displays a leading $ sign.	1234	$9999	‘ $1234’

Element	**Description**	*numeric-expression*	*format-value*	**Output**
C	Returns ISO currency symbol; USD. Restriction: You can specify the ISO currency symbol only at the beginning or end of the *format-value*.	12	C999	‘ USD12’
L	Returns local currency symbol Restriction: You can specify the local currency symbol only at the beginning or end of the *format-value*. The $ (dollar) is the only supported local currency symbol.	1234	L9999	‘ $1234’
MI	Returns leading or trailing minus sign(-). Restriction: You can specify the MI only at the beginning or end of the *format-value.*	-1234	9999MI	‘1234-’
PR	Returns negative value in <> brackets and positive value with leading and trailing blank spaces Restriction: You can specify the PR only at the end of the *format-value.*	-1234	9999PR	‘<1234>”
RN rn	Converts *numeric-expression* to Roman numerals in uppercase or lowercase. Restriction: You cannot include other elements with RN or rn.	485	RN rn	‘CDLXXXV’ ‘cdlxxxv’
S	Returns a leading or trailing plus or minus sign. Restriction: You can specify S element only at the beginning or end of the *format-value.*	-1245 1266	S9999 S9999	‘-1245’ ‘+1266’

Element	Description	*numeric-expression*	*format-value*	Output
U	Returns Euro or other dual currency symbol. Restriction: You can specify this symbol only at the end of the *format-value*. The $ (dollar) is the only supported currency symbol.	12	9999U	‘ $12’
V	Multiplies the specified *numeric-expression* by 10^n, where n specifies the number of 9s after V.	12	99V999	‘ 12000’
FM	Controls blank padding.	12	FM99	‘12’

Considerations

- The length of the numeric *format-value* must not exceed 128 characters.
- The *format-value* causes the *numeric-expression* to be truncated (not rounded) to the specified number of significant digits, for example:

  TO_CHAR(14.426, ‘99.99’) produces the truncated output value, 14.42.
- The elements D, FM, G, L, MI, PR, S, V, and U are case-insensitive.
- If the value has more significant digits to the left of the decimal point than in the specified *format-value* , the output is hash (#) sign, for example:

  TO_CHAR(1254, ‘999’) produces an output ####.
- All negative values have a leading negative sign (-) except when *format-value* contains MI, S, or PR elements, for example:
  - TO_CHAR(-1234, ‘9999’) produces an output of ‘-1234’. Notice the leading negative sign and the output does not contain MI, S, or PR
  - TO_CHAR(-1234, ‘9999S’) produces an output of ‘1234-’. Notice the negative sign at the end of the output for element S
  - TO_CHAR(-1234, ‘9999MI’) produces an output of ‘1234-’. Notice the negative sign at the end of the output for element MI
- The elements MI, S, C, and L can appear either at the beginning or end of the *format-value*, for example:
  - TO_CHAR(12, ‘C99’) has a valid *format-value*
  - TO_CHAR(12,‘9C9’) has an invalid *format-value* as the element C is neither at the beginning nor at the end of the *format-value*

- All positive values have a leading space, for example:

  TO_CHAR(12, ‘99’) has a leading space in the output, ‘ 12’.

- V multiplies the input values by 10^n, where n is the number of 9s following V. For example, if the input is 12 and the *format-value* is 99V999, the output is ‘ 12000’.

- The PR element must appear at the end of the *format-value* , for example:
  - TO_CHAR(-12, ‘99PR’) has a valid *format-value*
  - TO_CHAR(-12, ‘PR99’) has an invalid *format-value* as the element PR does not appear at the end of the *format-value*

- The elements S, MI, or PR must precede the elements C, L, or U at the beginning and must succeed at the end in the *format-value* , for example:
  - TO_CHAR(12, ‘SC99’) or TO_CHAR(12, ‘99CS’) has a valid *format-value*
  - TO_CHAR(12, ‘CS9’) or TO_CHAR(12, ‘99SC’) has an invalid *format-value*

- The RN element converts numbers from 1 through 3999 to roman numerals. All other values will have the output of # sign, for example:
  - TO_CHAR(12, ‘RN’) produces the output ‘XII’
  - TO_CHAR(-12, ‘RN’) produces the output '###############'
  - TO_CHAR(2585, ‘rn’) produces the output ‘mmdlxxxv’

- A FM (Fill-Mode) modifier can be applied to a *format-value* to control blank padding, for example:

  TO_CHAR(32, ‘FM9999’) has the output, ‘32’ with no leading space.

Examples of TO_CHAR(<NUMERIC>)

Expression	**Result**
TO_CHAR(-12.24, '99.99MI')	‘12.24-’
TO_CHAR(250, 'RN')	‘CCL’
TO_CHAR(25, 'FM009999')	‘000025’
TO_CHAR(-25, 'FM009999PR')	‘<000025>’
TO_CHAR(156.25, 'FML9999.99')	‘$156.25’
TO_CHAR(156.25, 'S9999.99')	‘+156.25’
TO_CHAR(148.5, '999D999S')	‘148.500+’
TO_CHAR(12, '99V999')	‘ 12000’
TO_CHAR(12.4, '99V999')	‘ 12400’
TO_CHAR(12.54, '99V9')	‘ 125’

TO_CHAR(<DATETIME>) Function

The `TO_CHAR(<DATETIME>)`function converts datetime data to formatted string. The result string is VARCHAR type.

The following is the function syntax:

```
TO_CHAR(datetime-expression, format-value)
```

datetime-expression

is an expression of datetime value which can be `DATE` or `TIMESTAMP`.NULL or dynamic parameter cannot be used for *datetime-expressions*.

format-value

is a constant string that defines the output format. NULL, column name, expression, or dynamic parameter cannot be specified as `format-value` . The following table lists all the datetime elements that can be used to construct the datetime `format-values`.

Element	**Description**	*datetime-expression*	*format-value*	**Output**
- / , . ; : “text”	Returns datetime field separator or punctuation in the output format. The text inside the quotation marks (“ ”) is reproduced without any change after removing the quotes.	31-MAR-11 05.02.31.123457 AM	‘DD/MM/YYYY’	‘31/03/2011’
AD A.D. BC B.C.	Returns Anno Domini or Before Christ indicator with or without periods.	31-MAR-11 05.02.31.123457 AM	‘YYYY AD’	‘2011 AD’
AM A.M. PM P.M.	Returns Ante Meridian or Post Meridian indicator with or without periods.	31-MAR-11 05.02.31.123457 AM	‘HH:M1 AM’	‘05:02 AM’
CC SCC	Returns Century	31-MAR-11 05.02.31.123457 AM	‘CC’	‘21’
D	Returns the day of week (1-7) Sunday = 1	31-MAR-11 05.02.31.123457 AM	‘D’	‘5’
DAY day Day	Returns the name of day in uppercase, lower case, or capitalized based on the case of the element, padded with blanks to display width of the widest name of the day.	31-MAR-11 05.02.31.123457 AM	‘DAY’ ‘day’ ’Day’	‘THURSDAY’ ‘thursday’ ‘Thursday’
DD	Returns the day (date) of month (1-31)	31-MAR-11 05.02.31.123457 AM	‘DD’	‘31’
DDD	Returns the day of year (1-366)	31-MAR-11 05.02.31.123457 AM	‘DDD’	‘090’

Element	Description	*datetime-expression*	*format-value*	Output
DY dy Dy	Returns the abbreviated name of the day in uppercase, lowercase, or capitalized based on case of the element.	31-MAR-11 05.02.31.123457 AM	‘DY’ ‘dy’ ‘Dy’	‘THU’ ‘thu’ ‘Thu’
FF[1..9]	Returns fractional seconds; the numbers 1 through 9 can be used after FF to specify the number of digits in fractional seconds. The default value is 6.	31-MAR-11 05.02.31.123457 AM	‘FF’ ‘FF2’ ‘FF4’	‘123457’ ‘12’ ‘1234’
FM	Fill-mode modifier is used to remove leading or trailing blanks.	31-MAR-11 05.02.31.123457 AM	‘FMMONTH’	‘MARCH’
HH HH12	Returns the hour of the day (1-12)	31-MAR-11 05.02.31.123457 AM	‘HH’	‘05’
HH24	Returns the hour of the day (0-23)	31-MAR-11 05.02.31.123457 AM	‘HH24’	‘05’
IW	Returns the week of the year (1-52 or 1-53) based on the ISO standard.	31-MAR-11 05.02.31.123457 AM	‘IW’	‘13’
IYYY IYY IY I	Returns last 4, 3, 2 or 1 digits of the ISO year.	31-MAR-11 05.02.31.123457 AM	‘IYYY’	‘011’
YEAR year Year SYEAR Syear SYear	Spell out year in uppercase, lowercase, or capitalized based on the case of the element.	31-MAR-11 05.02.31.123457 AM	‘YEAR’ ‘year’ ‘Year’	‘TWENTY ELEVEN ‘twenty eleven’ ‘Twenty Eleven’

Element	Description	*datetime-expression*	*format-value*	Output
J	Returns Julian day; the number of days since January 1, 4712 BC.	31-MAR-11 05.02.31.123457 AM	‘J’	‘2455652’
MI	Returns minutes (0-59)	31-MAR-11 05.02.31.123457 AM	‘MI’	‘02’
MM	Returns month (01-12; January = 01)	31-MAR-11 05.02.31.123457 AM	‘MM’	‘03’
MON mon Mon	Returns the abbreviated name of month in uppercase, lowercase, or capitalized based on the case of the element.	31-MAR-11 05.02.31.123457 AM	‘MON’ ‘mon’ ‘Mon’	‘MAR’ ‘mar’ ‘Mar’
MONTH month Month	Returns the name of month month in uppercase, lowercase, or capitalized based on the case of the element, padded with blanks to display width of the widest name of a month.	31-MAR-11 05.02.31.123457 AM	‘MONTH’ ‘month’ ‘Month’	‘MARCH’ ‘march’ ‘March
RM	Returns the month in roman numerals (I to XII).	31-MAR-11 05.02.31.123457 AM	‘RM’	‘III’
RR RRRR	Returns the last two or four digits of the year.	31-MAR-11 05.02.31.123457 AM	‘RR’ ‘RRRR’	‘11’ ‘2011’
SS	Returns seconds (0-59)	31-MAR-11 05.02.31.123457 AM	‘SS’	‘31’
SSSSS	Returns seconds past midnight (0-86399).	31-MAR-11 05.02.31.123457 AM	‘SSSSS’	‘18151’
X	Returns the local radix character. The ‘.’ (dot) is the only supported local radix character.	31-MAR-11 05.02.31.123457 AM	‘HH:MM:SSXFF’	‘05:03:31.123457’

Element	**Description**	*datetime-expression*	*format-value*	**Output**
Y,YYY	Returns the year with a comma at the specified position.	31-MAR-11 05.02.31.123457 AM	‘Y,YYY’	‘2,011’
YYYY SYYYY	Returns four digits year; S prefixes BC dates with a minus sign.	31-MAR-11 05.02.31.123457 AM	‘YYYY’	‘2011’
YYY YY Y	Returns the last three digits; two digits; or last digit of the year.	31-MAR-11 05.02.31.123457 AM	‘YY’	‘11’
WW	Returns the week of the year (1-53)	31-MAR-11 05.02.31.123457 AM	‘WW’	‘13’
W	Returns the week of the month (1-5)	31-MAR-11 05.02.31.123457 AM	‘W’	‘5’
th	Adds the suffix st/nd/th to the numeric output format. Note: The ‘th’ suffix can be added to any element; the suffix st/nd/th will be ignored for non-numeric output value.	31-MAR-11 05.02.31.123457 AM	‘DDth’ ‘DYth’	‘31st’ ‘THU’

Considerations

- The length of the datetime *format-value* must not exceed 74 characters.
- For elements whose output can be produced in uppercase, lowercase, or capitalized (the first character in uppercase and rest in lowercase), the output is based on the case of the element, for example:
  - TO_CHAR(timestamp‘2011-03-31 05.02.31.123457’, ‘DAY’) produces an output ‘THURSDAY ’.
  - TO_CHAR(timestamp‘2011-03-31 05.02.31.123457’, ‘day’) produces an output ‘thursday ’.
  - TO_CHAR(timestamp‘2011-03-31 05.02.31.123457, ‘Day’) produces an output ‘Thursday ’.
- The space character is used to fill the output value of an element to a constant width equal to the largest element for the relevant *format-value* , for example: WEDNESDAY has the maximum number of characters. Therefore, all the DAY

output formats will include 9 characters. If the output value is Tuesday, two space characters are padded to the right (‘TUESDAY ’).

- Fill Mode (FM) modifiers are used to control blank padding, for example: TO_CHAR(timestamp‘2011-03-31 05.02.31.123457’, ‘FMDay’) produces the output ‘THURSDAY’.
- The elements CC, DDD, DD, D, HH24, HH12, HH, IW, IYYY, IYY, IY, I, J, MI, MM, SSSSS, SS, WW, W, Y, YYYY, YYY, YY, Y, SCC, FF[1..9], RRRR, RR, SYYYY, and X are case-insensitive.
- The suffix element ‘th’ is ignored for non-numeric output, for example:
    - TO_CHAR(timestamp‘2011-09-28 15:15:10.599494’, ‘DDth’) produces 28th as output.
    - TO_CHAR(timestamp‘2011-09-28 15:15:10.599494’, ‘Dayth’) produces ‘Wednesday’ as output. Notice that ‘th’ is ignored.

Examples of TO_CHAR(<DATETIME>)

Expression	**Result**
TO_CHAR(timestamp‘2011-09-28 15:15:10.599494’, ‘Day, DD HH12:MI:SS’)	‘Wednesday, 28 03:15:10’
TO_CHAR(timestamp‘2011-09-28 15:15:10.599494’, ‘“Day:”Day’)	‘Day:Wednesday’
TO_CHAR(timestamp‘2011-09-28 15:15:10.599494’, ‘DAY’)	‘WEDNESDAY’
TO_CHAR(timestamp‘2011-09-28 15:15:10.599494’, ‘DD-MM-YY’)	‘28-09-2011’
TO_CHAR(timestamp‘2011-09-28 15:15:10.599494’, ‘HH:MI AM’)	‘03:15 PM’

TRANSLATE Function

The TRANSLATE function translates a character string from a source character set to a target character set.

```
TRANSLATE(character-value-expression USING translation-name)
```

character-value-expression

is a character string.

translation-name

is one of these translation names:

Translation Name	Source Character Set	Target Character Set	Comments
ISO8859XToUCS2 (X in 1)	ISO8859X	UCS2	No data loss is possible.
UCS2ToISO8859X (X in 1)	UCS2	ISO8859X	No data loss is possible. NonStop SQL/MX will display error 8413 if it encounters a Unicode character that cannot be converted to the target character set.
KANJITOISO88591	KANJI	ISO88591	Convert a KANJI source to a multibyte ISO88591 target. Every character is copied as is. No check on the source data. No data loss is possible.
KSC5601TOISO88591	KSC5601	ISO88591	Convert a KSC5601 source to a multibyte ISO88591 target. Every character is copied as is. No check on the source data. No data loss is possible.

translation name identifies the translation, source and target character set. When you translate to the UCS2 character set no data loss is possible. However, when NonStop SQL/MX translates a *character-value-expression* from UCS2, certain characters encoded in UTF16 cannot be converted to the target character set. NonStop SQL/MX displays an error in this case.

NonStop SQL/MX returns a variable-length character string with character repertoire equal to the character repertoire of the target character set of the translation and the maximum length equal to the fix length or maximum variable length of the source *character-value-expression*.

If you enter an illegal *translation-name*, NonStop SQL/MX returns an error.

If the character set for *character-value-expression* is different from the source character set as specified in the *translation-name*, NonStop SQL/MX returns an error.

TRIM Function

The TRIM function removes the specified characters from the left side, the right side, or both sides of a character string.

```
TRIM ([[trim-type] [trim-chrs] FROM] srcstr)

trim-type is:
   LEADING | TRAILING | BOTH
```

trim-type

specifies whether characters are to be trimmed from the left side (LEADING), right side (TRAILING), or both sides (BOTH) of *srcstr*. If you omit the *trim-type*, the default is BOTH.

trim_chrs

is the character or characters to be removed from the *srcstr*. The default is the space character.

srcstr

is a SQL character value expression from which spaces or trim characters are removed. See Character Value Expressions on page 6-41.

Considerations for TRIM

Result of TRIM

The result is always of type NON ANSI VARCHAR, with maximum length equal to the fixed length or maximum variable length of *srcstr*. If the source character string is an upshifts CHAR or VARCHAR string, the result is an upshifts VARCHAR type.

Examples of TRIM

- Return `'Robert'`:

```
TRIM ('    Robert    ')
```

- The EMPLOYEE table defines FIRST_NAME as CHAR(15) and LAST_NAME as CHAR(20). This expression uses the TRIM function to return the value `'Robert Smith'` without extra blanks:

```
TRIM (first_name) || ' ' || TRIM (last_name)
```

- In this example, characters '1' and '0' are removed from both sides of the `colA` value:

```
create table tab1(colA varchar(20), colB int);
insert into tab1 values('0101010101ten0101010101', 10);
```

```
insert into tab1 values('11001101nine11001101', 9);
select trim(BOTH '10' from cola), colb from tab01;

(EXPR)                COLB

--------------------  -----------

ten                            10

nine                            9
```

UCASE Function

Considerations for UCASE
Examples of UCASE

The UCASE function upshifts characters. UCASE can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the UCASE function is equal to the result returned by the UPPER or UPSHIFT function.

For UCS2 character-expression, the UCASE function upshifts all lowercase or title case characters to uppercase and returns a character string. If the argument is of type CHAR(n) or VARCHAR(n), the result is of type VARCHAR(min(3n, 2048)).

A lowercase character is a character that has the “alphabetic” property in Unicode Standard 2 and whose Unicode name includes *lower*. An upper case character is a character that has the “alphabetic” property and whose Unicode name includes *upper*. A title case character is a character that has the Unicode “alphabetic” property and whose Unicode name includes *title*.

UCASE returns a string of either fixed-length or variable-length character data, depending on the data type of the input string.

You cannot use the UCASE function on KANJI or KSC5601 operands.

UCASE is an SQL/MX extension.

```
UCASE (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to upshift. See Character Value Expressions on page 6-41.

Considerations for UCASE

Table 8-4 lists all one-to-one mappings for the UCS2 character set. In addition, it is possible for the result string to be longer than that of the source because some of the title case characters can be mapped to multiple characters.

Table 8-5 lists UCS2 characters with two-character uppercase mapping.

Table 8-6 lists UCS2 characters with three-character uppercase mapping.

Characters not listed in these tables use themselves as their uppercase mappings.

Table 8-4. One-to-One UCS2 Mappings (page 1 of 4)

x	U (x)	x	U(x)	x	U(x)	x	U(x)	x	U(x)	x	U(x)
0061	0041	0173	0172	03C9	03A9	04D5	04D4	1E75	1E74	1F72	1FC8
0062	0042	0175	0174	03CA	03AA	04D7	04D6	1E77	1E76	1F73	1FC9
0063	0043	0177	0176	03CB	03AB	04D9	04D8	1E79	1E78	1F74	1FCA
0064	0044	017A	0179	03CC	038C	04DB	04DA	1E7B	1E7A	1F75	1FCB
0065	0045	017C	017B	03CD	038E	04DD	04DC	1E7D	1E7C	1F76	1FDA
0066	0046	017E	017D	03CE	038F	04DF	04DE	1E7F	1E7E	1F77	1FDB
0067	0047	017F	0053	03D0	0392	04E1	04E0	1E81	1E80	1F78	1FF8
0068	0048	0183	0182	03D1	0398	04E3	04E2	1E83	1E82	1F79	1FF9
0069	0049	0185	0184	03D5	03A6	04E5	04E4	1E85	1E84	1F7A	1FEA
006A	004A	0188	0187	03D6	03A0	04E7	04E6	1E87	1E86	1F7B	1FEB
006B	004B	018C	018B	03E3	03E2	04E9	04E8	1E89	1E88	1F7C	1FFA
006C	004C	0192	0191	03E5	03E4	04EB	04EA	1E8B	1E8A	1F7D	1FFB
006D	004D	0199	0198	03E7	03E6	04EF	04EE	1E8D	1E8C	1F80	1F88
006E	004E	01A1	01A0	03E9	03E8	04F1	04F0	1E8F	1E8E	1F81	1F89
006F	004F	01A3	01A2	03EB	03EA	04F3	04F2	1E91	1E90	1F82	1F8A
0070	0050	01A5	01A4	03ED	03EC	04F5	04F4	1E93	1E92	1F83	1F8B
0071	0051	01A8	01A7	03EF	03EE	04F9	04F8	1E95	1E94	1F84	1F8C
0072	0052	01AD	01AC	03F0	039A	0561	0531	1E9B	1E60	1F85	1F8D
0073	0053	01B0	01AF	03F1	03A1	0562	0532	1EA1	1EA0	1F86	1F8E
0074	0054	01B4	01B3	03F2	03A3	0563	0533	1EA3	1EA2	1F87	1F8F
0075	0055	01B6	01B5	0430	0410	0564	0534	1EA5	1EA4	1F90	1F98
0076	0056	01B9	01B8	0431	0411	0565	0535	1EA7	1EA6	1F91	1F99
0077	0057	01BD	01BC	0432	0412	0566	0536	1EA9	1EA8	1F92	1F9A
0078	0058	01C5	01C4	0433	0413	0567	0537	1EAB	1EAA	1F93	1F9B
0079	0059	01C6	01C4	0434	0414	0568	0538	1EAD	1EAC	1F94	1F9C
007A	005A	01C8	01C7	0435	0415	0569	0539	1EAF	1EAE	1F95	1F9D
00E0	00C0	01C9	01C7	0436	0416	056A	053A	1EB1	1EB0	1F96	1F9E
00E1	00C1	01CB	01CA	0437	0417	056B	053B	1EB3	1EB2	1F97	1F9F
00E2	00C2	01CC	01CA	0438	0418	056C	053C	1EB5	1EB4	1FA0	1FA8
00E3	00C3	01CE	01CD	0439	0419	056D	053D	1EB7	1EB6	1FA1	1FA9
00E4	00C4	01D0	01CF	043A	041A	056E	053E	1EB9	1EB8	1FA2	1FAA
00E5	00C5	01D2	01D1	043B	041B	056F	053F	1EBB	1EBA	1FA3	1FAB
00E6	00C6	01D4	01D3	043C	041C	0570	0540	1EBD	1EBC	1FA4	1FAC
00E7	00C7	01D6	01D5	043D	041D	0571	0541	1EBF	1EBE	1FA5	1FAD

Table 8-4. One-to-One UCS2 Mappings (page 2 of 4)

x	U (x)	x	U(x)	x	U(x)	x	U(x)	x	U(x)	x	U(x)
00E8	00C8	01D8	01D7	043E	041E	0572	0542	1EC1	1EC0	1FA6	1FAE
00E9	00C9	01DA	01D9	043F	041F	0573	0543	1EC3	1EC2	1FA7	1FAF
00EA	00CA	01DC	01DB	0440	0420	0574	0544	1EC5	1EC4	1FB0	1FB8
00EB	00CB	01DD	018E	0441	0421	0575	0545	1EC7	1EC6	1FB1	1FB9
00EC	00CC	01DF	01DE	0442	0422	0576	0546	1EC9	1EC8	1FB3	1FBC
00ED	00CD	01E1	01E0	0443	0423	0577	0547	1ECB	1ECA	1FBE	0399
00EE	00CE	01E3	01E2	0444	0424	0578	0548	1ECD	1ECC	1FC3	1FCC
00EF	00CF	01E5	01E4	0445	0425	0579	0549	1ECF	1ECE	1FD0	1FD8
00F0	00D0	01E7	01E6	0446	0426	057A	054A	1ED1	1ED0	1FD1	1FD9
00F1	00D1	01E9	01E8	0447	0427	057B	054B	1ED3	1ED2	1FE0	1FE8
00F2	00D2	01EB	01EA	0448	0428	057C	054C	1ED5	1ED4	1FE1	1FE9
00F3	00D3	01ED	01EC	0449	0429	057D	054D	1ED7	1ED6	1FE5	1FEC
00F4	00D4	01EF	01EE	044A	042A	057E	054E	1ED9	1ED8	1FF3	1FFC
00F5	00D5	01F2	01F1	044B	042B	057F	054F	1EDB	1EDA	2170	2160
00F6	00D6	01F3	01F1	044C	042C	0580	0550	1EDD	1EDC	2171	2161
00F8	00D8	01F5	01F4	044D	042D	0581	0551	1EDF	1EDE	2172	2162
00F9	00D9	01FB	01FA	044E	042E	0582	0552	1EE1	1EE0	2173	2163
00FA	00DA	01FD	01FC	044F	042F	0583	0553	1EE3	1EE2	2174	2164
00FB	00DB	01FF	01FE	0451	0401	0584	0554	1EE5	1EE4	2175	2165
00FC	00DC	0201	0200	0452	0402	0585	0555	1EE7	1EE6	2176	2166
00FD	00DD	0203	0202	0453	0403	0586	0556	1EE9	1EE8	2177	2167
00FE	00DE	0205	0204	0454	0404	1E01	1E00	1EEB	1EEA	2178	2168
00FF	0178	0207	0206	0455	0405	1E03	1E02	1EED	1EEC	2179	2169
0101	0100	0209	0208	0456	0406	1E05	1E04	1EEF	1EEE	217A	216A
0103	0102	020B	020A	0457	0407	1E07	1E06	1EF1	1EF0	217B	216B
0105	0104	020D	020C	0458	0408	1E09	1E08	1EF3	1EF2	217C	216C
0107	0106	020F	020E	0459	0409	1E0B	1E0A	1EF5	1EF4	217D	216D
0109	0108	0211	0210	045A	040A	1E0D	1E0C	1EF7	1EF6	217E	216E
010B	010A	0213	0212	045B	040B	1E0F	1E0E	1EF9	1EF8	217F	216F
010D	010C	0215	0214	045C	040C	1E11	1E10	1F00	1F08	24D0	24B6
010F	010E	0217	0216	045E	040E	1E13	1E12	1F01	1F09	24D1	24B7
0111	0110	0253	0181	045F	040F	1E15	1E14	1F02	1F0A	24D2	24B8
0113	0112	0254	0186	0461	0460	1E17	1E16	1F03	1F0B	24D3	24B9
0115	0114	0256	0189	0463	0462	1E19	1E18	1F04	1F0C	24D4	24BA
0117	0116	0257	018A	0465	0464	1E1B	1E1A	1F05	1F0D	24D5	24BB
0119	0118	0259	018F	0467	0466	1E1D	1E1C	1F06	1F0E	24D6	24BC
011B	011A	025B	0190	0469	0468	1E1F	1E1E	1F07	1F0F	24D7	24BD
011D	011C	0260	0193	046B	046A	1E21	1E20	1F10	1F18	24D8	24BE
011F	011E	0263	0194	046D	046C	1E23	1E22	1F11	1F19	24D9	24BF

Table 8-4. One-to-One UCS2 Mappings (page 3 of 4)

x	U (x)	x	U(x)	x	U(x)	x	U(x)	x	U(x)	x	U(x)
0121	0120	0268	0197	046F	046E	1E25	1E24	1F12	1F1A	24DA	24C0
0123	0122	0269	0196	0471	0470	1E27	1E26	1F13	1F1B	24DB	24C1
0125	0124	026F	019C	0473	0472	1E29	1E28	1F14	1F1C	24DC	24C2
0127	0126	0272	019D	0475	0474	1E2B	1E2A	1F15	1F1D	24DD	24C3
0129	0128	0275	019F	0477	0476	1E2D	1E2C	1F20	1F28	24DE	24C4
012B	012A	0280	01A6	0479	0478	1E2F	1E2E	1F21	1F29	24DF	24C5
012D	012C	0283	01A9	047B	047A	1E31	1E30	1F22	1F2A	24E0	24C6
012F	012E	0288	01AE	047D	047C	1E33	1E32	1F23	1F2B	24E1	24C7
0131	0049	028A	01B1	047F	047E	1E35	1E34	1F24	1F2C	24E2	24C8
0133	0132	028B	01B2	0481	0480	1E37	1E36	1F25	1F2D	24E3	24C9
0135	0134	0292	01B7	0491	0490	1E39	1E38	1F26	1F2E	24E4	24CA
0137	0136	0345	0399	0493	0492	1E3B	1E3A	1F27	1F2F	24E5	24CB
013A	0139	03AC	0386	0495	0494	1E3D	1E3C	1F30	1F38	24E6	24CC
013C	013B	03AD	0388	0497	0496	1E3F	1E3E	1F31	1F39	24E7	24CD
013E	013D	03AE	0389	0499	0498	1E41	1E40	1F32	1F3A	24E8	24CE
0140	013F	03AF	038A	049B	049A	1E43	1E42	1F33	1F3B	24E9	24CF
0142	0141	03B1	0391	049D	049C	1E45	1E44	1F34	1F3C	FF41	FF21
0144	0143	03B2	0392	049F	049E	1E47	1E46	1F35	1F3D	FF42	FF22
0146	0145	03B3	0393	04A1	04A0	1E49	1E48	1F36	1F3E	FF43	FF23
0148	0147	03B4	0394	04A3	04A2	1E4B	1E4A	1F37	1F3F	FF44	FF24
014B	014A	03B5	0395	04A5	04A4	1E4D	1E4C	1F40	1F48	FF45	FF25
014D	014C	03B6	0396	04A7	04A6	1E4F	1E4E	1F41	1F49	FF46	FF26
014F	014E	03B7	0397	04A9	04A8	1E51	1E50	1F42	1F4A	FF47	FF27
0151	0150	03B8	0398	04AB	04AA	1E53	1E52	1F43	1F4B	FF48	FF28
0153	0152	03B9	0399	04AD	04AC	1E55	1E54	1F44	1F4C	FF49	FF29
0155	0154	03BA	039A	04AF	04AE	1E57	1E56	1F45	1F4D	FF4A	FF2A
0157	0156	03BB	039B	04B1	04B0	1E59	1E58	1F51	1F59	FF4B	FF2B
0159	0158	03BC	039C	04B3	04B2	1E5B	1E5A	1F53	1F5B	FF4C	FF2C
015B	015A	03BD	039D	04B5	04B4	1E5D	1E5C	1F55	1F5D	FF4D	FF2D
015D	015C	03BE	039E	04B7	04B6	1E5F	1E5E	1F57	1F5F	FF4E	FF2E
015F	015E	03BF	039F	04B9	04B8	1E61	1E60	1F60	1F68	FF4F	FF2F
0161	0160	03C0	03A0	04BB	04BA	1E63	1E62	1F61	1F69	FF50	FF30
0163	0162	03C1	03A1	04BD	04BC	1E65	1E64	1F62	1F6A	FF51	FF31
0165	0164	03C2	03A3	04BF	04BE	1E67	1E66	1F63	1F6B	FF52	FF32
0167	0166	03C3	03A3	04C2	04C1	1E69	1E68	1F64	1F6C	FF53	FF33
0169	0168	03C4	03A4	04C4	04C3	1E6B	1E6A	1F65	1F6D	FF54	FF34
016B	016A	03C5	03A5	04C8	04C7	1E6D	1E6C	1F66	1F6E	FF55	FF35
016D	016C	03C6	03A6	04CC	04CB	1E6F	1E6E	1F67	1F6F	FF56	FF36
016F	016E	03C7	03A7	04D1	04D0	1E71	1E70	1F70	1FBA	FF57	FF37

Table 8-4. One-to-One UCS2 Mappings (page 4 of 4)

x	U (x)	x	U(x)	x	U(x)	x	U(x)	x	U(x)	x	U(x)
0171	0170	03C8	03A8	04D3	04D2	1E73	1E72	1F71	1FBB	FF58	FF38
										FF59	FF39
										FF5A	FF3A

Table 8-5. Two-Character UCS2 Mapping (page 1 of 3)

Titlecase characters	Two-character uppercase expansions
0x00DF	0x0053 0x0053
0x0149	0x02BC 0x004E
0x01F0	0x004A 0x030C
0x0587	0x0535 0x0552
0x1E96	0x0048 0x0331
0x1E97	0x0054 0x0308
0x1E98	0x0057 0x030A
0x1E99	0x0059 0x030A
0x1E9A	0x0041 0x02BE
0x1F50	0x03A5 0x0313
0x1F80	0x1F08 0x0399
0x1F81	0x1F09 0x0399
0x1F82	0x1F0A 0x0399
0x1F83	0x1F0B 0x0399
0x1F84	0x1F0C 0x0399
0x1F85	0x1F0D 0x0399
0x1F86	0x1F0E 0x0399
0x1F87	0x1F0F 0x0399
0x1F88	0x1F08 0x0399
0x1F89	0x1F09 0x0399
0x1F8A	0x1F0A 0x0399
0x1F8B	0x1F0B 0x0399
0x1F8C	0x1F0C 0x0399
0x1F8D	0x1F0D 0x0399
0x1F8E	0x1F0E 0x0399
0x1F8F	0x1F0F 0x0399
0x1F90	0x1F28 0x0399
0x1F91	0x1F29 0x0399

Table 8-5. Two-Character UCS2 Mapping (page 2 of 3)

Titlecase characters	Two-character uppercase expansions
0x1F92	0x1F2A 0x0399
0x1F93	0x1F2B 0x0399
0x1F94	0x1F2C 0x0399
0x1F95	0x1F2D 0x0399
0x1F96	0x1F2E 0x0399
0x1F97	0x1F2F 0x0399
0x1F98	0x1F28 0x0399
0x1F99	0x1F29 0x0399
0x1F9A	0x1F2A 0x0399
0x1F9B	0x1F2B 0x0399
0x1F9C	0x1F2C 0x0399
0x1F9D	0x1F2D 0x0399
0x1F9E	0x1F2E 0x0399
0x1F9F	0x1F2F 0x0399
0x1FA0	0x1F68 0x0399
0x1FA1	0x1F69 0x0399
0x1FA2	0x1F6A 0x0399
0x1FA3	0x1F6B 0x0399
0x1FA4	0x1F6C 0x0399
0x1FA5	0x1F6D 0x0399
0x1FA6	0x1F6E 0x0399
0x1FA7	0x1F6F 0x0399
0x1FA8	0x1F68 0x0399
0x1FA9	0x1F69 0x0399
0x1FAA	0x1F6A 0x0399
0x1FAB	0x1F6B 0x0399
0x1FAC	0x1F6C 0x0399
0x1FAD	0x1F6D 0x0399
0x1FAE	0x1F6E 0x0399
0x1FAF	0x1F6F 0x0399
0x1FB2	0x1FBA 0x0399
0x1FB3	0x0391 0x0399
0x1FB4	0x0386 0x0399
0x1FB6	0x0391 0x0342

Table 8-5. Two-Character UCS2 Mapping (page 3 of 3)

Titlecase characters	Two-character uppercase expansions
0x1FBC	0x0391 0x0399
0x1FC2	0x1FCA 0x0399
0x1FC3	0x0397 0x0399
0x1FC4	0x0389 0x0399
0x1FC6	0x0397 0x0342
0x1FCC	0x0397 0x0399
0x1FD6	0x0399 0x0342
0x1FE4	0x03A1 0x0313
0x1FE6	0x03A5 0x0342
0x1FF2	0x1FFA 0x0399
0x1FF3	0x03A9 0x0399
0x1FF4	0x038F 0x0399
0x1FF6	0x03A9 0x0342
0x1FFC	0x03A9 0x0399
0xFB00	0x0046 0x0046
0xFB01	0x0046 0x0049
0xFB02	0x0046 0x004C
0xFB05	0x0053 0x0054
0xFB06	0x0053 0x0054
0xFB13	0x0544 0x0546
0xFB14	0x0544 0x0535
0xFB15	0x0544 0x053B
0xFB16	0x054E 0x0546
0xFB17	0x0544 0x053D

Table 8-6. Three-Character UCS2 Mapping (page 1 of 2)

Titlecase characters	Three-Character Uppercase Expansions
0x0390	0x0399 0x0308 0x0301
0x03B0	0x03A5 0x0308 0x0301
0x1F52	0x03A5 0x0313 0x0300
0x1F54	0x03A5 0x0313 0x0301
0x1F56	0x03A5 0x0313 0x0342
0x1FB7	0x0391 0x0342 0x0399

Table 8-6. Three-Character UCS2 Mapping (page 2 of 2)

Titlecase characters	Three-Character Uppercase Expansions
0x1FC7	0x0397 0x0342 0x0399
0x1FD2	0x0399 0x0308 0x0300
0x1FD3	0x0399 0x0308 0x0301
0x1FD7	0x0399 0x0308 0x0342
0x1FE2	0x03A5 0x0308 0x0300
0x1FE3	0x03A5 0x0308 0x0301
0x1FE7	0x03A5 0x0308 0x0342
0x1FF7	0x03A9 0x0342 0x0399
0xFB03	0x0046 0x0046 0x0049

Examples of UCASE

- Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return in uppercase and lowercase letters by using the UCASE and LCASE functions:

  ```
  SELECT custname,UCASE(custname),LCASE(custname)
  FROM sales.customer;
  ```

  ```
  (EXPR)                (EXPR)                 (EXPR)
  -----------------     -------------------    ------------------
  ...                   ...                    ...
  Hotel Oregon          HOTEL OREGON           hotel oregon

  --- 17 row(s) selected.
  ```

  See LCASE Function on page 8-87.

  For more examples of when to use the UCASE function, see UPSHIFT Function on page 8-202.

UPPER Function

The UPPER function upshifts characters. UPPER can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the UPPER function is equal to the result returned by the UPSHIFT or UCASE function.

UPPER returns a string of either fixed-length or variable-length character data, depending on the data type of the input string.

You cannot use the UPPER function on KANJI or KSC5601 operands.

```
UPPER (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to upshift. See Character Value Expressions on page 6-41.

Examples of UPPER

- Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return in uppercase and lowercase letters by using the UPPER and LOWER functions:

```
SELECT custname,UPPER(custname),LOWER(custname)
FROM sales.customer;

(EXPR)               (EXPR)               (EXPR)
-----------------   -------------------  ------------------
...                 ...                  ...
Hotel Oregon        HOTEL OREGON         hotel oregon

--- 17 row(s) selected.
```

See LOWER Function on page 8-94.

For examples of when to use the UPPER function, see UPSHIFT Function on page 8-202.

UPSHIFT Function

The UPSHIFT function upshifts characters. UPSHIFT can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the UPSHIFT function is equal to the result returned by the UPPER or UCASE function.

UPSHIFT returns a string of either fixed-length or variable-length character data, depending on the data type of the input string.

You cannot use the UPSHIFT function on KANJI or KSC5601 operands.

UPSHIFT is an SQL/MX extension.

```
UPSHIFT (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to upshift. See Character Value Expressions on page 6-41.

Examples of UPSHIFT

- Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return a result in uppercase and lowercase letters by using the UPSHIFT, UPPER, and LOWER functions:

```
SELECT UPSHIFT(custname),UPPER(custname),UCASE(custname)
FROM sales.customer;

(EXPR)             (EXPR)               (EXPR)
-----------------  -------------------  ------------------
...                ...                  ...
HOTEL OREGON       HOTEL OREGON         HOTEL OREGON

--- 17 row(s) selected.
```

- Perform a case-insensitive search for the DataSpeed customer:

```
SELECT *
FROM sales.customer
WHERE UPSHIFT (custname) = 'DATASPEED';

CUSTNUM  CUSTNAME    STREET                CITY       ...
-------  ----------  --------------------  ---------  ...
   1234  DataSpeed   300 SAN GABRIEL WAY   NEW YORK   ...

--- 1 row(s) selected.
```

In the table, the name can be in lowercase, uppercase, or mixed case letters.

- Suppose that your database includes two department tables: DEPT1 and DEPT2. Return all rows from the two tables in which the department names have the same value regardless of case:

```
SELECT * FROM persnl.dept1 D1, persnl.dept2 D2
WHERE UPSHIFT(D1.deptname) = UPSHIFT(D2.deptname);
```

USER Function

The USER function returns the current Guardian user ID as variable-length character data in the form *group.name*.

```
USER
```

The USER function is equivalent to the CURRENT_USER Function on page 8-43 and the SESSION_USER Function on page 8-168.

Examples of USER

- Retrieve the user name value for the current user:

```
SELECT USER FROM logfile;

(EXPR)
-----------------------------------------
DCS.TSHAW
...

--- 5 row(s) selected.
```

VERSION_INFO Function

VERSION_INFO is a built-in table-valued function that returns version information for a single entity.

```
version_info ('E_TYPE', 'E_VALUE')
```

Table 8-7 shows the input and output parameters for VERSION_INFO.

Table 8-7. Input and Output Parameters for VERSION_INFO

Input/Output Type	Parameter	Specification	Description
Input parameter	E_TYPE	CHAR (32) NOT NULL	The type of version information that is requested.
Input parameter	E_VALUE	VARCHAR(776) NOT NULL	The name of the entity for which version information is requested. The type of that entity is implied by E_TYPE.
Output column	E_TYPE	CHAR (32) NOT NULL	A copy of the actual value for the E_TYPE input parameter.
Output column	E_VALUE	VARCHAR(776) NOT NULL	A copy of the actual value for the E_VALUE input parameter.
Output column	VERSION	INT NOT NULL	The version of the specified entity.
Output column	NODE_NAME	CHAR(8) NOT NULL	The Expand node name of a node where the named entity is defined.
Output column	MXV	INT NOT NULL	The SQL/MX Software Version (MXV) of the Expand node specified by NODE_NAME. The artificial value 999999 indicates that the node is unavailable and the MXV could not be obtained. In that case, warning 25420 (`node could not be accessed`) is also returned, once per node that is unavailable. These warnings are returned when the cursor to read the result set is opened. Individual fetch operations do not return warning 25420.

Table 8-8 specifies the valid values for the E_TYPE and E_VALUE parameters. For all E_TYPE values, the NODE_NAME and MXV specify Expand node name and MXV of a node that is related to the corresponding entity.

Table 8-8. Values for the E_TYPE and E_VALUE Parameters

Value for the E_TYPE and E_VALUE Parameters	Description
SYSTEM_SCHEMA	The specified node.
SCHEMA	All nodes where the catalog of the schema is visible. By default, this includes the local node.
TABLE	All nodes where partitions of that table reside, and the local node.
TABLE_ALL	All nodes where partitions of that table reside.
	All nodes where partitions of indexes on that table reside.
	All nodes where partitions of an associated trigger temp table reside, and the local node.
INDEX	All nodes where partitions of that index reside and the local node.
INDEX_TABLE	The union of TABLE for the base table of the index and INDEX for the index itself.
VIEW	All nodes where replicas for that view reside and the local node.
PROCEDURE	All nodes where replicas for that procedure reside and the local node.
MPALIAS	The node of the target SQL/MP partition and the local node.
CONSTRAINT, MODULE, and TRIGGER	The local node.

Table 8-8 shows the VERSION output column values for the E_TYPE and E_VALUE parameters.

Table 8-9. VERSION Output Column Values E_TYPE and E_VALUE Parameters (page 1 of 2)

E_TYPE	E_VALUE	VERSION
SYSTEM_SCHEMA	Expand node name. Use local node if spaces.	System schema version for actual node
SCHEMA	ANSI name of schema or database object	Schema version
TABLE	ANSI name of table	OFV of table
TABLE_ALL		
INDEX	ANSI name of index	OFV of index
INDEX_TABLE		
VIEW	ANSI name of view	OFV of view

Table 8-9. VERSION Output Column Values E_TYPE and E_VALUE Parameters (page 2 of 2)

E_TYPE	E_VALUE	VERSION
CONSTRAINT	ANSI name of constraint	OFV of constraint
TRIGGER	ANSI name of trigger	OFV of trigger
PROCEDURE	ANSI name of procedure	OFV of procedure
MPALIAS	ANSI name of mpalias	OFV of mpalias
MODULE	ANSI name of module	Module version

ANSI names in the input value parameter must be fully qualified, in external format. Expand node names are case-insensitive. Both input parameters must be character-valued expressions.

Example of VERSION_INFO

```
select * from table (version_info ('SCHEMA', 'CAT.SCH'));

E_TYPE      E_VALUE       VERSION   NODE_NAME   MXV___
-------     --------      -------   ---------   ----
SCHEMA      CAT.SCH       1200      \REMOTE     1200
SCHEMA      CAT.SCH       1200      \XYZZY      1400
    ...
```

VARIANCE Function

Considerations for VARIANCE
Examples of VARIANCE

VARIANCE is an aggregate function that returns the statistical variance of a set of numbers.

VARIANCE is an SQL/MX extension.

```
VARIANCE ([ALL | DISTINCT] expression [,weight])
```

`ALL | DISTINCT`

specifies whether duplicate values are included in the computation of the VARIANCE of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the VARIANCE function is applied. If DISTINCT is specified, you cannot specify *weight*.

expression

specifies a numeric value expression that determines the values for which to compute the variance. *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the VARIANCE function operates on distinct values from the one-column table derived from the evaluation of *expression*.

weight

specifies a numeric value expression that determines the weights of the values for which to compute the variance. *weight* cannot contain an aggregate function or a subquery. *weight* is defined on the same table as *expression*. The one-column table derived from the evaluation of *expression* and the one-column table derived from the evaluation of *weight* must have the same cardinality.

Considerations for VARIANCE

Definition of VARIANCE

Suppose that v_i are the values in the one-column table derived from the evaluation of *expression*. *N* is the cardinality of this one-column table that is the result of applying the *expression* to each row of the source table and eliminating rows that are null.

If *weight* is specified, w_i are the values derived from the evaluation of *weight*. *N* is the cardinality of the two-column table that is the result of applying the *expression* and *weight* to each row of the source table and eliminating rows that have nulls in either column.

Definition When Weight Is Not Specified

If *weight* is not specified, the statistical variance of the values in the one-column result table is defined as:

$$\frac{\sum_{i=1}^{N}(v_i - \overline{v})^2}{N - 1}$$

where v_i is the *i*-th value of *expression*, $\overline{v}$ is the average value expressed in the common data type, and *N* is the cardinality of the result table.

Because the definition of variance has $N-1$ in the denominator of the expression (when weight is not specified), NonStop SQL/MX returns a default value of zero (and no error) if the number of rows in the table, or a group of the table, is equal to 1.

Definition When Weight Is Specified

If *weight* is specified, the statistical variance of the values in the two-column result table is defined as:

$$\frac{\sum_{i=1}^{N}(v_i - \overline{vw})^2 \cdot w_i}{\sum_{i=1}^{N} w_i - 1}$$

where v_i is the *i*-th value of *expression*, w_i is the *i*-th value of *weight*, $\overline{vw}$ is the weighted average value expressed in the common data type, and *N* is the cardinality of the result table.

Weighted Average

The weighted average $\overline{vw}$ of v_i and w_i is defined as:

$$\frac{\sum_{i=1}^{N} v_i \cdot w_i}{\sum_{i=1}^{N} w_i}$$

where v_i is the i-th value of *expression*, w_i is the i-th value of *weight*, and N is the cardinality of the result table.

Data Type of the Result

The data type of the result is always DOUBLE PRECISION.

Operands of the Expression

The expression includes columns from the rows of the SELECT result table—but cannot include an aggregate function. These expressions are valid:

```
VARIANCE (SALARY)
VARIANCE (SALARY * 1.1)
VARIANCE (PARTCOST * QTY_ORDERED)
```

Nulls

VARIANCE is evaluated after eliminating all nulls from the set. If the result table is empty, VARIANCE returns NULL.

FLOAT(54) and DOUBLE PRECISION Data

Avoid using large FLOAT(54) or DOUBLE PRECISION values as arguments to VARIANCE. If SUM(x * x) exceeds the value of 1.15792089237316192e77 during the computation of VARIANCE(x), then a numeric overflow occurs.

Examples of VARIANCE

- Compute the variance of the salary of the current employees:

```
SELECT VARIANCE(salary) AS Variance_Salary
FROM persnl.employee;

VARIANCE_SALARY
-------------------------
 1.27573263588496116E+009

--- 1 row(s) selected.
```

- Compute the variance of the cost of parts in the current inventory:

```
SELECT VARIANCE (price * qty_available)
FROM sales.parts;

(EXPR)
-------------------------
 5.09652410092950336E+013

--- 1 row(s) selected.
```

- Suppose that your database includes a WEATHER table, which is created by using SQLCI in this way:

```
CREATE TABLE $db.mining.weather
 ( city          VARCHAR (20) NO DEFAULT NOT NULL
  ,state         CHAR (2)     NO DEFAULT NOT NULL
  ,date_weather DATE NO DEFAULT NOT NULL
  ,temperature  NUMERIC (3) SIGNED
  ,weight       NUMERIC (2) UNSIGNED
  ,PRIMARY KEY (city, state, date_weather))
  CATALOG $db.mining
  ORGANIZATION KEY SEQUENCED;
```

After the table is created, you can insert the mapping into the OBJECTS table by using MXCI in this way:

```
CREATE SQLMP ALIAS db.mining.weather $db.mining.weather;
```

For these examples, the WEATHER table contains these rows:

CITY	STATE	DATE_WEATHER	TEMPERATURE	WEIGHT
Austin	TX	1997-01-01	50	1
Austin	TX	1997-01-02	40	1
Austin	TX	1997-01-03	60	2
Austin	TX	1997-01-04	84	2
Cupertino	CA	1997-01-01	65	1
Cupertino	CA	1997-01-02	65	2
Cupertino	CA	1997-01-03	65	2
Cupertino	CA	1997-01-04	65	1

- Find the variance of the TEMPERATURE column:

```
SELECT VARIANCE(temperature) FROM weather;

(EXPR)
-------------------------
 1.64500000000000024E+002

--- 1 row(s) selected.
```

- Find the variance of the TEMPERATURE column by CITY:

```
SELECT city, VARIANCE (temperature)
  FROM weather GROUP BY city;

CITY        (EXPR)
----------  -------------------------
Austin                3.55666666666666720E+002
Cupertino             0.00000000000000000E+000

--- 2 row(s) selected.
```

- Find the weighted variance of the TEMPERATURE column:

```
SELECT VARIANCE (temperature, weight) FROM weather;

(EXPR)
-------------------------
 1.46363636363636384E+002

--- 1 row(s) selected.
```

WEEK Function

The WEEK function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 54 that represents the corresponding week of the year.

WEEK is an SQL/MX extension.

```
WEEK (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of WEEK

- Return an integer that represents the week of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, WEEK(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  --------------
1996-04-10  1996-04-21 08:15:00.000000              15
```

YEAR Function

The YEAR function converts a DATE or TIMESTAMP expression into an INTEGER value that represents the year.

YEAR is an SQL/MX extension.

```
YEAR (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See Datetime Value Expressions on page 6-43.

Examples of YEAR

- Return an integer that represents the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, YEAR(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
1996-04-10  1996-04-21 08:15:00.000000    1996
```

9 SQL/MX File Attributes

The ATTRIBUTE file option describes physical characteristics of files (including files that contain tables, indexes, or partitions) that can affect the performance of applications that use the files.

File attributes are set when a file (or an object that resides in a file) is created. If you do not specify ATTRIBUTE values in the statement that creates an object (such as CREATE TABLE or CREATE INDEX), NonStop SQL/MX uses default values for file attributes.

Many file attributes can be changed later (with statements such as ALTER TABLE or ALTER INDEX), some file attributes remain in effect for the life of the object, and a few file attributes can change as a side effect of a command or a change to some other attribute.

This section describes file attributes for SQL/MX objects:

ALLOCATE/DEALLOCATE on page 9-2	Reserves or frees disk space for a file.
AUDITCOMPRESS on page 9-3	Controls whether unchanged columns are included in audit records.
BLOCKSIZE on page 9-4	Sets size of data blocks. Default is 4096.
CLEARONPURGE on page 9-5	Controls disk erasure when file is dropped.
EXTENT on page 9-6	Controls the size of extents.
MAXEXTENTS on page 9-7	Controls the number of extents.

For more information, see the separate entry for a specific attribute.

ALLOCATE/DEALLOCATE

ALLOCATE is a Guardian file attribute that reserves disk space for a file. DEALLOCATE frees disk space previously reserved for the file that does not contain data. ALLOCATE applies to tables and indexes. Allocate disk space in advance to ensure that space is available when needed and to avoid processing errors caused by full or fragmented disks during normal allocation-on-demand.

You set the ALLOCATE attribute for a table with CREATE TABLE or ALTER TABLE. You set the attribute for an index with CREATE INDEX or ALTER INDEX. You set the DEALLOCATE attribute with ALTER TABLE or ALTER INDEX.

```
ALLOCATE num-extents | DEALLOCATE
```

The default is ALLOCATE 0. You cannot allocate extents during table or index creation unless you specify the ALLOCATE attribute.

`ALLOCATE` *`num-extents`*

is the number of extents to allocate in advance. The number must be an integer between 0 and the current value of the MAXEXTENTS file attribute. If the object contains partitions, the number of extents to allocate cannot be greater than the MAXEXTENTS partition attribute for any of the partitions.

Depending on your file configuration, you might not be able to allocate the full number of MAXEXTENTS. For ALTER TABLE or ALTER INDEX, ALLOCATE allocates new extents until the total of new and existing extents equals the specified number.

`DEALLOCATE`

frees all unused allocated extents (that is, all allocated extents beyond the extent that contains the end-of-file).

Considerations for ALLOCATE

ALLOCATE and DEALLOCATE apply to all partitions of the specified file.

ALLOCATE affects the number of extents, but not the size of the extents. The EXTENT file attribute determines the extent size.

If the number of extents to allocate is less than or equal to the current number of extents allocated, the ALLOCATE operation does nothing. To decrease the number of extents allocated, you must perform a DEALLOCATE operation to deallocate any unused extents, followed by an ALLOCATE operation to allocate the desired number of extents.

AUDITCOMPRESS

The AUDITCOMPRESS file attribute controls whether TMF audit records for the file are compressed. Compressed audit records omit unchanged columns from the before and after images of updated rows. Uncompressed audit records allow you to read complete rows in the audit trail but require more space.

You set the AUDITCOMPRESS attribute for a table with CREATE TABLE or ALTER TABLE. You set the AUDITCOMPRESS attribute for an index with CREATE INDEX or ALTER INDEX.

```
AUDITCOMPRESS | NO AUDITCOMPRESS
```

The table default is AUDITCOMPRESS. The index default is the table value at index creation.

Considerations for AUDITCOMPRESS

Index Default

By default, the AUDITCOMPRESS attribute is automatically set for an index to match that of the associated base table. If the AUDITCOMPRESS file attribute is changed on the base table, that change is automatically propagated to the index.

Difference Between Compressed and Uncompressed Row Images

Audit records of uncompressed files contain entire before and after images of changed rows. Audit records of compressed files generally contain only changed columns and columns of the clustering key. Other columns are occasionally included to improve performance, such as when a single unchanged column physically occurs between several changed columns.

BLOCKSIZE

The BLOCKSIZE file attribute specifies the number of bytes in a block.

Set the BLOCKSIZE attribute for a table or index with CREATE TABLE or CREATE INDEX statements. You cannot change the BLOCKSIZE attribute of an existing table or index.

```
BLOCKSIZE number-bytes
```

The default is BLOCKSIZE 4096.

number-bytes

is an integer that specifies the number of bytes in a block. Block size must be 4096 or 32768 bytes. If you specify a different block size, an error is returned.

For information on large block support and rollback of large block, see the *SQL/MX Installation and Management Guide*.

CLEARONPURGE

The CLEARONPURGE file attribute controls erasure of data from the disk when a file is deleted.

You set the CLEARONPURGE attribute for an SQL/MX table with CREATE TABLE or ALTER TABLE. You set the CLEARONPURGE attribute for an SQL/MX index with CREATE INDEX or ALTER INDEX.

```
CLEARONPURGE | NO CLEARONPURGE
```

The table default is NO CLEARONPURGE. The index default for the CLEARONPURGE attribute is the table value at index creation. If the CLEARONPURGE file attribute is changed on the base table, that change is automatically propagated to the index.

Considerations for CLEARONPURGE

Purpose of CLEARONPURGE

When you drop or purge a table or index with NO CLEARONPURGE, NonStop SQL/MX deallocates disk space but does not physically destroy the data in that disk space. This implementation improves performance by reducing writes to the disk, but when the disk space is allocated to a new file, other users might be able to read data left by the object that used the space previously.

CLEARONPURGE increases security for sensitive data by causing the system to overwrite deallocated disk space.

Effect Within Transactions

If you drop or purge a file with the CLEARONPURGE attribute from within a TMF transaction, the data is not physically erased from the disk until after the transaction commits.

EXTENT

EXTENT is a Guardian file attribute that sets the size of the extents (units of contiguous disk space) that will be allocated for a file or partition of a file. EXTENT applies to tables and indexes and is set when a file or partition is created.

```
EXTENT ext-size  | (pri-ext-size [, sec-ext-size ])

ext-size, pri-ext-size, sec-ext-size is:
  integer [PAGE[S] ]
```

The default is 16 pages for the primary extent and 64 pages for each secondary extent. A page consists of 2048 bytes.

integer

is an integer that specifies the number of pages in the extent. The ranges allowed are from 0 pages to the number of pages that will fit on a disk. The only limit is the physical amount of storage available.

Each partition of a partitioned file has its own EXTENT attribute that can differ from the EXTENT attribute for other partitions of the file. You can specify a single EXTENT size for each extent in the file or partition, or you can specify one size for the primary (first) extent and another size for the secondary extents.

If you enter only one value, with or without parentheses, it will be used for both the primary and secondary extent sizes. If you enter two values, they will be used for primary and secondary extent sizes.

Considerations for EXTENT

- A file's extent size must be at least as large as its block size and must be a multiple of the block size and a multiple of page size (2048 bytes). If you specify extent sizes that do not meet these conditions, NonStop SQL/MX uses the next block size or the next full page size. For example, 0 PAGE rounds up to 2 PAGEs.

  A file (or partition of a partitioned file) must fit on a disk, so the size of the primary extent plus the total size of all secondary extents must not exceed the disk size.

- A primary extent should be large enough to hold the file at the initial load, and secondary extents should be large enough to accommodate growth. The faster the growth, the larger the secondary extents should be.

  To ensure adequate space for your file, choose extent sizes and a MAXEXTENTS value large enough to accommodate the amount of data you expect to store in the file.

  Using large extents can improve performance by reducing the number of seeks. The disadvantage of large extents is that an entire extent is allocated simultaneously, leaving allocated but unused space on the disk while the extent contains only a small amount of data. You can maximize the use of disk space by specifying smaller extent sizes if performance is not an issue.

MAXEXTENTS

MAXEXTENTS is a file attribute that specifies the maximum number of extents that you can allocate for an unpartitioned file or for each partition of a partitioned file. MAXEXTENTS applies to tables and to indexes.

You set the MAXEXTENTS attribute for a table with CREATE TABLE or ALTER TABLE. You set the attribute for an index with CREATE INDEX or ALTER INDEX. You use the PARTITION clause with CREATE TABLE or CREATE INDEX to set the MAXEXTENTS attribute for a partition.

Unlike the NonStop SQL/MP form of these statements, the SQL/MX's ALTER TABLE and ALTER INDEX statements have no PARTONLY clause. When you supply a new value for attributes, these statements modify the value of the attribute on all partitions of the table or index.

```
MAXEXTENTS num-extents
```

num-extents

is an integer from 1 to 768 (but not less than the number of extents currently allocated for the file) that specifies the maximum number of extents that you can allocate. The default is 160.

Considerations for MAXEXTENTS

It is generally not efficient to have partitions with hundreds of extents, so keep MAXEXTENTS well below the allowed maximum value. If necessary, increase the number of partitions.

10 Metadata Tables

This section describes:

NonStop SQL/MX stores system metadata for all objects in SQL/MX tables, automatically creating and maintaining metadata as users create, alter, drop, or update statistics for SQL/MX objects.

User tables are the tables you typically create as a user. You can modify data in and alter or drop user tables, and you can grant privileges so that others can access and change data in your user tables.

NonStop SQL/MX user metadata for histograms is stored in HISTOGRAMS and HISTOGRAM_INTERVALS SQL/MX tables in each user schema. SQL/MP user metadata for histograms is stored in the SQL/MP tables HISTOGRM and HISTINTS. For more information about histograms, see User Metadata Tables (UMD): Histogram Tables on page 10-85.

User metadata that specifies system default settings for options when you execute SQL queries are stored in the SYSTEM_DEFAULTS table in schema SYSTEM_DEFAULTS_SCHEMA. You can modify data in user metadata tables and grant privileges on user metadata tables. You cannot alter or drop these tables.

System metadata about objects is stored in numerous tables in system schemas. You cannot modify data directly in the system metadata tables, but they are secured for PUBLIC SELECT access so that you can query them. The actual owner of the metadata schemas is an authorization ID specified at the time NonStop SQL/MX is installed.

In each of the table descriptions that follow:

- An asterisk preceding a column number indicates that the column is part of the clustering key, which is also called as primary key. Unless otherwise stated, the primary key is in column number sequence.
- Unless otherwise stated, timestamps are Julian timestamps.
- Unless otherwise stated, character data is stored in uppercase letters except for character columns that contain delimited identifiers, which are stored as is, but

without the surrounding double quotes, and with two consecutive double quotes collapsed into one double quote.

- Unless otherwise stated, CHAR(n) in the data type field without a character set qualifier is associated with the ISO88591 character set. All character types are searched or sorted with the DEFAULT (binary) collation with the PAD SPACE characteristic.

For SQL/MX Release 3.2.1, these tables are visible but are not supported and are reserved for future use:

- EXCEPTION_USAGE
- HISTOGRAM_FREQ_VALS
- MVGROUPS
- MVS
- MVS_COLS
- MVS_JOIN_COLS
- MVS_TABLE_INFO
- MVS_USED
- MVS_TABLE_INFO_UMD
- MVS_USED_UMD
- SCH_PRIVILEGES
- SYNONYM_USAGE

SQL/MX Metadata Catalogs

There is one system catalog per node where NonStop SQL/MX has been initialized. The system catalog name is NONSTOP_SQLMX_*nodename.* All metadata tables in the system catalog are stored on the volume specified as the system metadata volume during installation. There are six schemas in the system catalog:

- DEFINITION_SCHEMA_VERSION_*vernum*.
- MXCS_SCHEMA
- SYSTEM_DEFAULTS_SCHEMA
- SYSTEM_SCHEMA
- SYSTEM_SECURITY_SCHEMA
- SYSTEM_SQLJ_SCHEMA

You can create as many user catalogs as you wish on a node. Each will contain a DEFINITION_SCHEMA_VERSION_*vernum* schema.

SQL/MX Metadata Schemas and Tables

Note. See the diagrammatic representation of the SQL/MX metadata tables in the *SQL/MX Installation and Management Guide* that is applicable for SQL/MX Release 3.2.

System Schema Tables: Schema SYSTEM_SCHEMA

System metadata to resolve object names is stored in schema, SYSTEM_SCHEMA in the system catalog NONSTOP_SQLMX_*nodename*.

There is one system catalog per node where NonStop SQL/MX has been initialized.

This table lists the metadata tables in the system schema:

[1]ALL_UIDS Table on page 10-8	UIDs for all objects that have metadata on *node*
CATSYS Table on page 10-9	Catalogs visible from *node*
CAT_REFERENCES Table on page 10-9	Catalog reference information for catalogs visible from *node*
SCHEMATA Table on page 10-10	Schemas in catalog visible from *node*
SCHEMA_REPLICAS Table on page 10-11	Replica information for schemas with definitions on *node*

1. The ALL_UIDS table is present in version 1200 system schema only.

Definition Schema Tables: Schema DEFINITION_SCHEMA_VERSION_*vernum*

Additional system metadata for each object is stored in schema DEFINITION_SCHEMA_VERSION_*vernum* in the catalog that contains the object.

NonStop SQL/MX automatically creates this schema and all its tables when you execute the first CREATE SCHEMA statement for that catalog.

Within system metadata tables, each catalog, schema or object is identified by a unique ID (UID). A UID is a 64-bit number generated and assigned to the catalog, schema, or object at the time of creation. A catalog UID is unique within the set of nodes where that catalog is visible. A schema UID is unique among the set of schemas in the same catalog. An object UID is unique among the set of objects in the same catalog.

This table lists definition schema tables in schema DEFINITION_SCHEMA_VERSION_*vernum* of each catalog:

ACCESS_PATHS Table on page 10-12	Physical instances (a table, index, or a partition) of data in the catalog
ACCESS_PATH_COLS Table on page 10-14	Columns in physical instances of data
CK_COL_USAGE Table on page 10-15	Columns referenced in search conditions of check constraints
CK_TBL_USAGE Table on page 10-15	Tables referenced in search conditions of check constraints
COLS Table on page 10-15	Columns in tables and views
COL_PRIVILEGES Table on page 10-20	Grant information for columns
DDL_LOCKS Table on page 10-21	Lock information for controlling concurrent DDL operations on an object
DDL_PARTITION_LOCKS on page 10-21	DDL locks being held on partitions
KEY_COL_USAGE Table on page 10-22	Constraints on key columns
MP_PARTITIONS Table on page 10-22	Partition names of SQL/MP tables with SQL/MX aliases
OBJECTS Table on page 10-22	Tables, views, indexes, and constraints
PARTITIONS Table on page 10-24	Partitions in the catalog
REF_CONSTRAINTS Table on page 10-25	Referential constraints on tables in the catalog
REPLICAS Table on page 10-26	Location of replicas in the catalog
RI_UNIQUE_USAGE Table on page 10-26	Unique constraints and their referential constraints
ROUTINES Table on page 10-27	User-defined stored procedures
SEQUENCE_GENERATORS Table on page 10-28	Contains the Sequence Generator attributes.
SG_USAGE Table on page 10-29	Represents the usage of Sequence Generator objects by other objects
TBL_CONSTRAINTS Table on page 10-29	Constraints on a table
TBL_PRIVILEGES Table on page 10-30	Grant information for tables
TEXT Table on page 10-32	Text associated with objects
TRIGGERS Table on page 10-32	Information about triggers

TRIGGERS_CAT_USAGE Table on page 10-34	How triggers use objects in other catalogs
TRIGGER_USED Table on page 10-34	Describes how triggers use objects
VWS Table on page 10-35	Views in the catalog
VW_COL_TBL_COLS Table on page 10-36	Base table columns used in views
VW_COL_USAGE Table on page 10-36	Columns used in views
VW_TBL_USAGE Table on page 10-36	Tables referenced by views

System Defaults Tables (User Metadata Tables): Schema SYSTEM_DEFAULTS_SCHEMA

User metadata that specifies system default settings for options and other attributes when you execute SQL queries are stored in the SYSTEM_DEFAULTS table in the schema SYSTEM_DEFAULTS_SCHEMA of catalog NONSTOP_SQLMX_*nodename*.

The `InstallSqlmx` script automatically creates the SYSTEM_DEFAULTS table with the system catalog when you install NonStop SQL/MX. For more information, see the *SQL/MX Installation and Management Guide*. Although this user metadata table is in a schema in the system catalog, it is not a system metadata table but rather a user metadata table with the security of the user who installs NonStop SQL/MX (normally the Super ID).

This table lists system defaults tables (user metadata tables) in NONSTOP_SQLMX_*nodename*.SYSTEM_DEFAULTS_SCHEMA:

System Defaults Table on page 10-37	Default settings for system options for SQL queries and SQLCI commands run through MXCI or through an application.

MXCS Metadata Tables: Schema MXCS_SCHEMA

This table lists MXCS tables in the MXCS_SCHEMA:

ASSOC2DS Table on page 10-95	Associates MXCS service to a data source
DATASOURCES Table on page 10-96	Data source information
ENVIRONMENTVALUES Table on page 10-97	Sets, controls and defines environment values

NAME2ID Table on page 10-97	Associates service or data source name to ID
RESOURCEPOLICIES Table on page 10-98	Governing information

Histogram Tables

These tables contain histograms that show how data is distributed with respect to a column or a group of columns within a table. These statistics enable the optimizer to create efficient access plans.

SQL/MX HISTOGRAM_INTERVALS and HISTOGRAMS tables are created when a user schema is created. The UPDATE STATISTICS statement inserts data into these tables. These files are also called user metadata (UMD) tables.

This table lists SQL/MX user metadata tables (UMD) in each user schema**:**

HISTOGRAMS Table on page 10-87	Columns, interval count, total number of rows and unique rows, and the low and high values of column distribution for the table for which the histogram is created
HISTOGRAM_INTERVALS Table on page 10-89	For each interval of the table for which the histogram is created, the number of rows and unique rows and the value of the upper boundary

If you are using SQL/MP tables for your data, there are histogram tables on the SQL/MP system. SQL/MP HISTOGRM and HISTINTS tables are automatically created in the same user catalog as the primary partition of the table you specify when you run the SQL/MX UPDATE STATISTICS statement. They are kept in SQL/MP files. For more information about SQL/MP metadata, see the *SQL/MP Reference Manual*.

This table lists SQL/MP histogram tables:

HISTOGRM Table on page 10-90	Columns, interval count, total number of rows and unique rows, and the low and high values of column distribution for the table for which the histogram is created
HISTINTS Table on page 10-91	For each interval of the table for which the histogram is created, the number of rows and unique rows and the value of the interval upper boundary

For detailed descriptions of histogram tables, see User Metadata Tables (UMD): Histogram Tables on page 10-85.

VALIDATEROUTINE: Schema SYSTEM_SQLJ_SCHEMA

This schema contains two stored procedures, VALIDATEROUTINE and VALIDATEROUTINE2, which are for internal use.

Security Schema Tables: Schema SYSTEM_SECURITY_SCHEMA

The SYSTEM_SECURITY_SCHEMA contains system metadata that holds node-wide security related information.

There is one system security schema per node where NonStop SQL/MX has been initialized or upgraded with schema version 3100 or later.

This table lists system security schema tables in the NONSTOP_SQLMX_*nodename*.SYSTEM_SECURITY_SCHEMA:

MGM_PRIVILEGES on page 10-11	Contains information about system and catalog wide privileges.
PRIVILEGED_USERS TABLE on page 10-11	Contains information about the users who are members of security administrators group.

System Schema Tables

ALL_UIDS Table

ALL_UIDS is a metadata table in NONSTOP_SQLMX_*nodename*.SYSTEM_SCHEMA that lists UIDs for all objects that have metadata on the node:

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of object
2 SCHEMA_UID	LARGEINT	UID of schema; link to SCHEMATA
3 OBJECT_NAME	CHAR(128)	Simple object name
4 OBJECT_NAME_SPACE	CHAR(2)	Object namespace: CN Constraint IX Index LK Lock TA Table value object (table, view, stored procedure, SQL/MP alias) TR Trigger TT Trigger temp table

* Indicates primary key

Note. The ALL_UIDS table is not present in schema version 3000 and newer system schemas.

An ALL_UIDS table contains one row per UID that is present in an OBJECTS table on the local node. Object names that are regular identifiers are stored in uppercase letters. Object names that are delimited identifiers are stored as is, without surrounding quotation marks.

All other character columns store letters in uppercase.

CATSYS Table

CATSYS is a metadata table in NONSTOP_SQLMX_*nodename*.SYSTEM_SCHEMA that describes all catalogs visible from the node:

Column Name	Data Type	Description
*1 CAT_NAME	CHAR(128)	Catalog name
2 CAT_UID	LARGEINT	UID for catalog; link to SCHEMATA and CAT_REFERENCES
3 REPLICATION_RULE	CHAR(2)	A if automatic schema replication rule M if manual schema replication rule
4 LOCAL_SMD_VOLUME	CHAR(8)	Volume where SMD and UMD tables in this catalog reside on the local node, including leading “$” sign.
5 LOCAL_USER_SCHEMA_COUNT	INT	Reserved for future use
6 CAT_OWNER	INT	Catalog owner’s user ID

* Indicates primary key

Catalog names that are regular identifiers are stored in uppercase letters. Catalog names that are delimited identifiers are stored as is, without surrounding quotation marks.

All other character columns store letters in uppercase.

CAT_REFERENCES Table

CAT_REFERENCES is a metadata table in NONSTOP_SQLMX_*nodename*.SYSTEM_SCHEMA that describes the locations of catalog references for catalogs visible from the node:

Column Name	Data Type	Description
*1CAT_UID	LARGEINT	UID for catalog; link to CATSYS
*2 NODE_NAME	CHAR(8)	Expand node name of node where the catalog is visible, including leading “\” (backslash)
3 SMD_VOLUME	CHAR(8)	Volume where SMD tables reside on the node, including leading “$” (dollar sign)
4 REPLICATION_RULE	CHAR(2)	A if automatic schema replication rule M if manual schema replication rule

SCHEMATA Table

SCHEMATA is a metadata table in NONSTOP_SQLMX_*nodename*.SYSTEM_SCHEMA that lists all schemas in all catalogs that are visible on the node.

Column Name	Data Type	Description
*1 CAT_UID	LARGEINT	UID of catalog for schema; link to CATSYS
*2 SCHEMA_NAME	CHAR(128)	Schema name
3 SCHEMA_UID	LARGEINT	UID of schema; link to OBJECTS
4 SCHEMA_OWNER	INT	Owner's user ID
5 SCHEMA_VERSION	INT	Version of schema: 1200 for R2.x 3000 for R3.0 3100 for R3.1 3200 for R3.2
6 SCHEMA_SUBVOLUME	CHAR(8)	Name of Guardian subvolume where objects from schema are stored.
7 CURRENT_OPERATION	CHAR(2)	Specifies if a schema level operation is active for the schema. Possible values are: spaces (no operation) CO a change ownership operation DC a DOWNGRADE ALL METADATA IN CATALOG operation DG a DOWNGRADE ALL METADATA operation UC an UPGRADE ALL METADATA IN CATALOG operation UG an UPGRADE ALL METADATA operation
8 SOURCE_VERSION	INT	The version of the schema before the execution of the operation indicated by the CURRENT_OPERATION column. Possible values are: 0 (if no operation is in progress) 1200 for R2.0 3000 for R3.0 3100 for R3.1 3200 for R3.2
9 TARGET_VERSION	INT	The target version of the operation indicated by the CURRENT_OPERATION column. Possible values are: 0 (if no operation is in progress) 3000 for R3.0 1200 for R2.0 3100 for R3.1 3200 for R3.2

* Indicates primary key

Schema names that are regular identifiers are stored in uppercase letters. Schema names that are delimited identifiers are stored as is, without surrounding quotation marks.

SCHEMA_REPLICAS Table

SCHEMA_REPLICAS is a metadata table in NONSTOP_SQLMX_*nodename*.SYSTEM_SCHEMA that lists locations of all replicas for all schemas that have definitions on the node.

Column Name	Data Type	Description
*1 SCHEMA_UID	LARGEINT	UID of schema; link to SCHEMATA
*2 NODE_NAME	CHAR(8)	Expand node name of node where replica resides, including leading "\" (backslash)

* Indicates primary key

System Security Schema Tables

MGM_PRIVILEGES

The MGM_PRIVILEGES is a metadata table, introduced in version 3100, stores node-wide user privileges.

Column Name	Data Type	Description
1 GRANTOR	INT	Security ID of the grantor.
2 GRANTOR_TYPE	CHAR(2)	O if grantor is a catalog owner
		U if grantor is a Security Administrator
*3 GRANTEE	INT	Security ID of the grantee.
4 GRANTEE_TYPE	CHAR(2)	U if user grant.
*5 PRIVILEGE_TYPE	CHAR(2)	CC if Create Catalog
		CS if Create Schema
* 6 TARGET_UID	LARGEINT	-1 when the privilege type is CC
		UID of target catalog, when the privilege type is CS
7 GRANT_TIMESTAMP	LARGEINT	Specifies the time when the GRANTOR granted the privilege to the GRANTEE.
8 IS_GRANTABLE	CHAR(2)	Reserved for future use.

* Indicates primary key

PRIVILEGED_USERS TABLE

The PRIVILEGED_USERS metadata table, introduced in version 3100, consists of rows of GRANTEE user IDs that are designated as members of a privileged user class by the GRANTOR user ID. The timestamp (GRANT_TIME) indicates the date and time when the GRANTEE was granted the designation contained in the USER_CLASS column. The set of user IDs in the GRANTEE column can modify the PRIVILEGED_USERS table. If the table is empty, then the Super ID can modify the PRIVILEGED_USERS table.

Column Name	Data Type	Description
*1 GRANTEE	INT	User ID of the user belonging to the user class.
*2 GRANTEE_TYPE	CHAR(2)	U = User
*3 USER CLASS	CHAR(2)	Denotes the user class. SA = Security Administrator
4 GRANTOR	INT	User ID of the user that designated the GRANTEE as a member of the user class.
5 GRANTOR_TYPE	CHAR(2)	U = User
6 IS_GRANTABLE	CHAR(2)	N = Not grantable
7 GRANT_TIME	LARGEINT	Specifies the Julian timestamp when the GRANTOR designated (granted) the GRANTEE as a member of the user class. This attribute is useful for security auditing purposes.

* Indicates primary key

Definition Schema Tables

ACCESS_PATHS Table

ACCESS_PATHS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that describes physical instances of data in the catalog. Base tables and indexes have physical instances. The actual locations of the physical instances are described in the PARTITIONS table.

Column Name	Data Type	Description
*1 TABLE_UID	LARGEINT	UID of base table; link to OBJECTS
*2 ACCESS_PATH_UID	LARGEINT	If path is index, UID of index; otherwise, UID of base table; link to PARTITIONS
3 ACCESS_PATH_TYPE	CHAR(2)	BT Base Table IX Index
4 COLUMN_COUNT	INT	Number of rows in table ACCESS_PATH_COLUMNS directly associated with this access path

Column Name	Data Type	Description
5 UNIQUE_COLUMN_COUNT	INT	Number of rows in table ACCESS_PATH_COLUMNS in unique key for this access path
6 VALID_DATA	CHAR(2)	Y if valid data N if not
7 RECORD_SIZE	INT	Number of bytes in each logical record
8 UNIQUES	CHAR(2)	Y if each row in this access path is unique N if not
9 EXPLICIT	CHAR(2)	Y if user-created index N if not
10 CLUSTERING_SCHEME	CHAR(2)	Physical organization of this access path: KS if by key
11 PARTITIONING_SCHEME	CHAR(2)	Partitioning method for this access path: N Not partitioned RP Range partitioned by first key HP Hash-1 partitioned
12 BLOCK_SIZE	INT	Number of bytes for disk blocks on this access path
13 KEY_LENGTH	INT	Number of bytes in key
14 PARTITIONING_KEY_LENGTH	INT	Number of bytes in partitioning key
15 LOCK_LENGTH	INT	Reserved for future use
16 AUDITED	CHAR(2)	Y if this path is audited N if not
17 AUDIT_COMPRESS	CHAR(2)	Y if audit is compressed N if not
18 CLEAR_ON_PURGE	CHAR(2)	Y if deleted records are cleared N if not
19 BUFFERED	CHAR(2)	Reserved for future use
20 RECORD_PACKED	CHAR(2)	Reserved for future use
21 DATA_COMPRESSED	CHAR(2)	Reserved for future use
22 INDEX_COMPRESSED	CHAR(2)	Y if index blocks are compressed N if not
23 PACKING_SCHEME	INT	Reserved for future use
24 PACKING_FACTOR	INT	Reserved for future use
25 ALL_COLUMNS_INCLUDED	CHAR(2)	Y if all columns included N if not
26 ROW_FORMAT	CHAR(2)	Reserved for future use

Column Name	Data Type	Description
27 INSERT_MODE	CHAR(2)	Reserved for future use
28 MAX_TABLE_SIZE	INT	Reserved for future use
29 RESERVED_FILLER_INT	INT	Reserved for future use
30 RESERVED_FILLER_CHAR	CHAR(20)	Reserved for future use
31 DISK_POOL	INT	Reserved for future use
32 NUM_DISK_POOL	INT	Reserved for future use

* Indicates primary key

In version 1200 schemas, the primary key consists of the ACCESS_PATH_UID column. In version 3000 and higher version schemas, the primary key consists of the columns in the following order:

1. TABLE_UID
2. ACCESS_PATH_UID

ACCESS_PATH_COLS Table

ACCESS_PATH_COLS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that describes columns within each access paths for the catalog:

Column Name	Data Type	Description
*1 ACCESS_PATH_UID	LARGEINT	UID of access path
*2 POSITION_IN_ROW	INT	Ordinal of column within access path (first position is 0)
3 COLUMN_NUMBER	INT	Position within row of base table (first column is 0)
4 ORDERING	CHAR(2)	A if ascending order D if descending order
5 PART_KEY_SEQ_NUM	INT	Order in partitioning key (0 if not in key)
6 CLUSTERING_KEY_SEQ_NUM	INT	Order in clustering key (0 if not in key)
7 SYSTEM_ADDED_COLUMN	CHAR(2)	Y if system added the column N if user added the column

* Indicates primary key

CK_COL_USAGE Table

CK_COL_USAGE is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that lists columns referenced by search conditions of check constraints in the catalog.

Column Name	Data Type	Description
*1 CONSTRAINT_UID	LARGEINT	UID of constraint
*2 TABLE_UID	LARGEINT	UID of table with referenced column
*3 COLUMN_NUMBER	INT	Column position in table (first column is 0)
4 SELECTS	CHAR(2)	Y if column is subject of a SELECT query in constraint definition N if not

* Indicates primary key

CK_TBL_USAGE Table

CK_TBL_USAGE is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that lists tables referenced by search conditions of check constraints in the catalog.

Column Name	Data Type	Description
*1 CONSTRAINT_UID	LARGEINT	UID of constraint
*2 TABLE_UID	LARGEINT	UID of table with referenced column

* Indicates primary key

COLS Table

COLS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that describes columns in tables and views in the catalog. The COLS table also contains attributes of the individual parameters of an SPJ, one row per parameter.

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of table, view or stored procedure
*2 COLUMN_NUMBER	INT	Logical position within row (first column is 0)
3 DIRECTION	CHAR(2)	I if input parameter to stored procedure O if OUTPUT parameter to stored procedure N if INPUT/OUTPUT parameter to stored procedure
4 COLUMN_CLASS	CHAR(2)	S if system-defined U if user-defined A if user-defined added column
5 COLUMN_NAME	CHAR(128)	Column name
6 COLUMN_SIZE	LARGEINT	Data bytes in column

Column Name	Data Type	Description
7 SQL_DATA_TYPE	CHAR(18)	One of these SQL data types: CHARACTER DATE DATETIME SIGNED DECIMAL UNSIGNED DECIMAL DOUBLE FLOAT SIGNED INTEGER INTERVAL UNSIGNED BP INT UNSIGNED INTEGER SIGNED LARGEINT SIGNED NUMERIC UNSIGNED NUMERIC REAL SIGNED SMALLINT UNSIGNED SMALLINT TIME TIMESTAMP VARCHAR LONG VARCHAR
8 CHARACTER_SET	CHAR(40)	Character set
9 ENCODING	CHAR (40)	Internal representation of columns with character data types.
10 COLLATION_SEQUENCE	CHAR(40)	Collation

Column Name	Data Type	Description
11 FS_DATA_TYPE	INT	File system data type. Values are: 0 fixed length ASCII string 64 variable length ASCII string 66 variable length double byte CHAR 70 MXCS long VARCHAR 130 16 bit signed 131 16 bit unsigned 132 32 bit signed 133 32 bit unsigned 134 64 bit signed 142 32 bit floating-point (IEEE format) 143 64 bit floating-point (IEEE format) 150 unsigned decimal 152 leading sign embedded 155 unsigned Bignum (unsigned numeric > 18 digits> 156 signed Bignum (signed numeric > 18 digits) 195 years 196 months 197 years and months 198 days 199 hours 200 days and hours 201 minutes 202 hours and minutes 203 days, hours, and minutes 204 seconds 205 minutes and seconds 206 hours, minutes, and seconds 207 days, hours, minutes, and seconds 208 fractional seconds
12 COL_SCALE	INT	Scale if numeric; fractional seconds if datetime or INTERVAL
13 COL_PRECISION	INT	If numeric, number of digits If FLOAT, number of digits of binary precision If INTERVAL, first field
14 UPSHIFTED	CHAR(2)	Y if type is CHAR, VARCHAR or PIC with UPSHIFT clause N if not
15 NULL_HEADER_SIZE	INT	Length of NULL indicator header
16 VARLEN_HEADER_SIZE	INT	Length of VARCHAR header

Column Name	Data Type	Description
17 DEFAULT_CLASS	CHAR(2)	Default defined by: Blank No default CD Current default ND Null default UD User default IA Identity Always ID Identity by default If CD, the value of the column depends on its data type: DATE Current date TIME Current time TIMESTAMP Current timestamp
18 LOGGABLE	CHAR(2)	Reserved for future use
19 DATETIME_START_ FIELD	INT	First field if type datetime or INTERVAL: 1 if year 2 if month 3 if day 4 if hour 5 if minute 6 if second
20 DATETIME_END_FIELD	INT	Last field if type DATE or INTERVAL; 6 if TIME or TIMESTAMP
21 DATETIME_LEADING_ PRECISION	INT	Precision in digits of first field if INTERVAL
22 DATETIME_TRAILING_ PRECISION	INT	Number of digits in fraction (of a second) if TIME, TIMESTAMP, or INTERVAL when last field is second
23 DATETIME_QUALIFIER	VARCHAR(28)	If datetime, text for start and end fields; blank for other types
24 DEFAULT_VALUE	VARNCHAR (240) CHARACTER SET UCS2	Column default value. Stored as Unicode characters.
25 HEADING_TEXT	VARCHAR(128)	Heading for column.
26 PICTURE_TEXT	VARCHAR(64)	PIC text if defined by COBOL85 PIC; blank if datetime or real
27 COLUMN_VALUE_DRIFT_PER_DAY	VARCHAR(128)	Reserved for future use
28 DATE_DISPLAY_FORMAT	VARCHAR(64)	Reserved for future use
29 CASE_SENSITIVE_COMPARISON	CHAR(2)	Case-sensitive comparison, if character type column. Y case-sensitive N case-insensitive

Column Name	Data Type	Description
30 DISPLAY_DATA_TYPE	VARCHAR(128)	Reserved for future use
31 RESERVED_FILLER_INT	INT	Reserved for future use
32 RESERVED_FILLER_CHAR	CHAR(20)	Reserved for future use

* Indicates primary key

Column names that are regular identifiers are stored in uppercase letters. Column names that are delimited identifiers are stored as is, without surrounding quotation marks.

All other character columns except DEFAULT_VALUE and HEADING_TEXT store letters in uppercase.

An IDENTITY column is defined as a part of the DEFAULT clause of `column-definition`. Depending on the type of the IDENTITY column either of the following two values can be present in the `DEFAULT_CLASS` column:

- IA - Identity Always, system automatically generates the values. User specified values are not accepted.
- ID - Identity by Default, either the system automatically generates the values, or the user specifies a value.

COL_PRIVILEGES Table

COL_PRIVILEGES is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that stores grant information for columns in the catalog:

Column Name	Data Type	Description
*1 TABLE_UID	LARGEINT	UID of table
*2 COLUMN_NUMBER	INT	Position within table (first column is 0)
*3 GRANTOR	INT	Security ID of grantor (or of owner if grantor is super ID acting for owner)
*4 GRANTOR_TYPE	CHAR(2)	S if system grant U if user grant O if granted as schema owner
*5 GRANTEE	INT	If GRANTEE_TYPE is U, security ID of grantee and link to TABLE_PRIVILEGES; no meaning otherwise
*6 GRANTEE_TYPE	CHAR(2)	P if public grant U if user grant O if granted as schema owner
* 7 PRIVILEGE_TYPE	CHAR(2)	Privilege type: S SELECT I INSERT D DELETE U UPDATE R REFERENCES
8 IS_GRANTABLE	CHAR(2)	Y if granted with grant option N if not

* Indicates primary key

Grant information for all tables is stored separately in the TBL_PRIVILEGES table.

DDL_LOCKS Table

DDL_LOCKS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that is used for control locking of an object so that utility operations in progress are protected against conflicting DDL or utility operations:

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of lock
2 BASE_OBJECT_UID	LARGEINT	UID of locked object
3 TIME_LOCK_REQUESTED	LARGEINT	Lock creation time
4 TIME_LOCK_ALTERED	LARGEINT	Time when the lock was last altered
5 OPERATION	CHAR(2)	Utility operation requesting the lock: CR Concurrent Restore DP Dup IM Import MT Modify table MI Modify index PI Populate index PD PURGEDATA RC Recover RS Restore UM Upgrade all metadata FS FASTCOPY Source object FT FASTCOPY Target object DM Downgrade all metadata
6 STATUS	INT	Step in the operation
7 PERCENT_COMPLETE	INT	Reserved for future use
8 PROCESS_CREATE_TIME	LARGEINT	Time when process was created
9 PROCESS_ID	VARCHAR(100)	ID of process that requested the lock

* Indicates primary key

DDL_PARTITION_LOCKS

The DDL_PARTITION_LOCKS table stores information about DDL locks being held on partitions.

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of a lock.
*2 SYSTEM_NAME	CHAR(8)	Name of a node. The node name includes the leading '\' sign.
*3 DATA_SOURCE	CHAR(8)	Name of a volume. The volume name includes the leading '$' sign.
*4 FILE_SUFFIX	CHAR(18)	Subvol and simple name.

* Indicates primary key

KEY_COL_USAGE Table

KEY_COL_USAGE is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that lists columns on key constraints in the catalog. KEY_COL_USAGE contains one or more rows for each unique, primary key, or foreign key constraint in the TBL_CONSTRAINTS table:

Column Name	Data Type	Description
*1 CONSTRAINT_UID	LARGEINT	UID of constraint
*2 COLUMN_NUMBER	INT	Position within table (first column is 0)
3 ORDINAL_POSITION	INT	Position within key (first column is 0)

* Indicates primary key

MP_PARTITIONS Table

MP_PARTITIONS is a metadata table in DEFINITION_SCHEMA that stores partition names of SQL/MP tables that have SQL/MX aliases:

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of table; link to OBJECTS
2 MPPARTITION_NAME	CHAR(36)	Name of NonStop SQL/MP partition system.volume.subvolume.name

* Indicates primary key

OBJECTS Table

OBJECTS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that describes tables, views, indexes, constraints, triggers, MP aliases, stored procedures, locks, and trigger temporary tables.

Column Name	Data Type	Description
*1 SCHEMA_UID	LARGEINT	UID of schema; link to SCHEMATA
*2 OBJECT_NAME	CHAR(128)	Simple object name
*3 OBJECT_NAME_SPACE	CHAR(2)	Object namespace: CN Constraint IX Index LK Lock SG Internal Sequence Generator TA Table value object (table, view, stored procedure, SQL/MP alias) TR Trigger TT Trigger temp table

Column Name	Data Type	Description
4 OBJECT_TYPE	CHAR(2)	Object type: BT Base table CC Check constraint IX Index LK Lock MP SQL/MP alias PV SQL/MP alias to an MP protection view SV SQL/MP alias to an MP shorthand view NN Not null constraint PK Primary key constraint RC Referential constraint TR Trigger object UC Unique constraint UR User-defined routine (for example, a stored procedure) VI View
5 OBJECT_UID	LARGEINT	UID of object
6 CREATE_TIME	LARGEINT	Julian timestamp of creation time
7 REDEF_TIME	LARGEINT	Julian timestamp of redefinition time
8 CACHE_TIME	LARGEINT	Julian timestamp of cache time
9 OBJECT_FEATURE_VERSION	INT	Feature version of the object. Starting with SQL/MX Release 3.2, the value is 3200 for all objects that use 3200 features
10 VALID_DEF	CHAR(2)	Y if definition valid N if not
11 OBJECT_SECURITY_CLASS	CHAR(2)	UM User metadata table UT User-defined table SM System metadata table
12 OBJECT_OWNER	INT	The integer representation of the owner's authorization ID
13 RESERVED_FILLER_INT	INT	Reserved for future use
14 RESERVED_FILLER_CHAR	CHAR(20)	Reserved for future use
15 DROPPABLE	CHAR(2)	Reserved for future use
16 RCB_VERSION	INT	The version of the object's Record Control Block (RCB)

* Indicates primary key

Note. In the definition schema version 3000 and later versions, a unique index called OBJIDX is defined on the OBJECT_UID column.

Object names that are regular identifiers are stored in uppercase letters. Object names that are delimited identifiers are stored as is, without surrounding quotation marks.

PARTITIONS Table

PARTITIONS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that describes partitions in the catalog. The columns INDEX_LEVEL, NON_EMPTY_BLOCK_COUNT, and EOF are updated by the UPDATE STATISTICS statement.

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of partitioned object
*2 SYSTEM_NAME	CHAR(8)	Name of node with partition including leading “\” (backslash)
*3 DATA_SOURCE	CHAR(8)	Name of volume with partition including leading “$” (dollar sign)
*4 FILE_SUFFIX	CHAR(18)	Subvolume and simple name of name of file with partition
5 PARTITION_NAME	VARCHAR(128)	Name associated with a partition.
6 MAX_SIZE	LARGEINT	The value of MAXSIZE for the partition
7 PRI_EXT	LARGEINT	Primary extent size
8 SEC_EXT	LARGEINT	Secondary extent size
9 MAX_EXT	LARGEINT	Maximum extent size
10 INDEX_LEVEL	INT	Index level of last UPDATE STATISTICS
11 NON_EMPTY_BLOCK_COUNT	INT	Number of nonempty blocks at last UPDATE STATISTICS
12 EOF	LARGEINT	End of file indication at last UPDATE STATISTICS
13 PARTITION_STATUS	CHAR(2)	Status defined by: AV Available UO Unavailable, offline UC Unavailable, corrupt UD Unavailable, dropped UR Unavailable, re-created
14 DDL_IN_PROGRESS	CHAR(2)	Reserved for future use
15 FIRST_KEY	VARCHAR(28670)	User-specified first key in normalized form; otherwise, zero-length. If hash partitioned, a logical partition number.

Column Name	Data Type	Description
16 ENCODED_KEY	VARCHAR(2732)	Internal encoded value of first key for the partition
17 PARTITION_DROP_TIME	LARGEINT	Julian timestamp of when the partition was dropped.

* Indicates primary key

All character columns store letters in uppercase except PARTITION_NAME and FIRST_KEY.

REF_CONSTRAINTS Table

REF_CONSTRAINTS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that describes referential constraints on tables in the catalog. It links each referential constraint to a unique constraint for the referenced table.

Column Name	Data Type	Description
*1 CONSTRAINT_UID	LARGEINT	UID of constraint, link to TBL_CONSTRAINTS
2 UNIQUE_CONSTRAINT_CAT_UID	LARGEINT	UID of catalog with referenced unique constraint
3 UNIQUE_CONSTRAINT_SCH_UID	LARGEINT	UID of schema with referenced unique constraint
4 UNIQUE_CONSTRAINT_UID	LARGEINT	UID of referenced unique constraint
5 MATCH_OPTION	CHAR(2)	Reserved for future use
6 UPDATE_RULE	CHAR(2)	CA CASCADE RE RESTRICT NA NO ACTION SD SET DEFAULT SN SET NULL
7 DELETE_RULE	CHAR(2)	CA CASCADE RE RESTRICT NA NO ACTION SD SET DEFAULT SN SET NULL

* Indicates primary key

REPLICAS Table

REPLICAS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that stores the locations of views and stored procedures in the catalog.

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of view
*2 SYSTEM_NAME	CHAR(8)	Name of node with view label, including leading “\” (backslash)
*3 DATA_SOURCE	CHAR(8)	Name of volume with view label, including leading “$” (dollar sign)
*4 FILE_SUFFIX	CHAR(18)	Subvolume and simple name of name of file with view label

* Indicates primary key

SYSTEM_NAME, DATA_SOURCE, and FILE_SUFFIX are stored in uppercase letters.

RI_UNIQUE_USAGE Table

RI_UNIQUE_USAGE is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that links unique constraints in the catalog with referential constraints that reference them:

Column Name	Data Type	Description
*1 UNIQUE_CONSTRAINT_UID	LARGEINT	UID of unique constraint
*2 FOREIGN_KEY_CATALOG_UID	LARGEINT	UID of referencing catalog
*3 FOREIGN_KEY_SCHEMA_UID	LARGEINT	UID of referencing schema
*4 FOREIGN_KEY_UID	LARGEINT	UID of referential constraint

* Indicates primary key

ROUTINES Table

ROUTINES is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that contains SPJ-level attributes, one row for each stored procedure in Java (SPJ) created in this catalog.

VARCHAR columns store letters as is (not converted to uppercase).

Column Name	Data Type	Description
*1 UDR_UID	LARGEINT	UID of procedure object
2 UDR_TYPE	CHAR(2)	P for procedure
3 LANGUAGE_TYPE	CHAR(2)	J for Java
4 DETERMINISTIC_BOOL	CHAR(2)	Y if deterministic N if not
5 SQL_ACCESS	CHAR(2)	M=MODIFIES SQL DATA N=NO SQL C=CONTAINS SQL R=READS SQL DATA
6 CALL_ON_NULL	CHAR(2)	Y (call the SPJ if a parameter passed to it is null)
7 ISOLATE_BOOL	CHAR(2)	Y (run in separate process)
8 PARAM_STYLE	CHAR(2)	J for Java
9 EXTRA_CALL	CHAR(2)	N (no extra calls)
10 TRANSACTION_ATTRIBUTES	CHAR(2)	Always RQ (Reserved for future use)
11 MAX_RESULTS	INT	Positive values in the range 0–255 appear with SPJ result sets
12 STATE_AREA_SIZE	INT	Reserved for future use
13 UDR_ATTRIBUTES	VARCHAR(128)	Reserved for future use
14 EXTERNAL_PATH	VARCHAR(256)	Value of specified EXTERNAL PATH
15 EXTERNAL_FILE	VARCHAR(256)	Name of the Java class, possibly prefixed by a package name
16 EXTERNAL_NAME	VARCHAR(128)	Simple name of the Java method

* Indicates primary key

SEQUENCE_GENERATORS Table

SEQUENCE_GENERATORS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that contains sequence generator attributes.

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of sequence generator object
2 SG_TYPE	CHAR(2)	Sequence generator type:
		I - Internal sequence generator
		E - External sequence generator
**3 START_VALUE	NUMERIC(128,0)	Sequence generator start value
**4 INCREMENT	NUMERIC(128,0)	Increment value for the sequence generator
5 SQL_DATA_TYPE	CHAR(18)	SQL data type, same as for the COLS table
6 FS_DATA_TYPE	INTEGER	File system data type, same as for the COLS table
**7 MAX_VALUE	NUMERIC(128,0)	The maximum value for the sequence generator
**8 MIN_VALUE	NUMERIC(128,0)	The minimum value for the sequence generator
9 CYCLE_OPTION	CHAR(2)	Y/N indication if the CYCLE option is used
10 CACHE	NUMERIC(128,0)	Reserved for future use
11 SG_ORDER	CHAR(2)	Reserved for future use
12 EXTENDED_PRECISION	INTEGER	Numeric precision, used internally when generating sequence numbers. The possible values are in the range 18-28

* Indicates primary key

** In versions prior to SQL/MX Release 3.2, the data type is LARGEINT

SG_USAGE Table

SG_USAGE is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that represents the usage of Sequence Generator objects by other objects.

Column number	Column Name	Data Type	Description
*1	SG_OBJECT_UID	LARGEINT	Object UID of used SG object
*2	USING_OBJECT_OBJ_UID	LARGEINT	Object UID of using object
*3	SG_CAT_UID	LARGEINT	Catalog UID of used SG object
*4	USING_OBJECT_CAT_UID	LARGEINT	Catalog UID of using object
5	SG_SCH_UID	LARGEINT	Schema UID of used SG object
6	USING_OBJECT_SCH_UID	LARGEINT	Schema UID of using object

* Indicates primary key

Each SG_USAGE row links a base table with an IDENTITY column to the associated internal Sequence Generator.

TBL_CONSTRAINTS Table

TBL_CONSTRAINTS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that contains one entry for each unique, primary key, foreign, or check constraint on a table in the catalog:

Column Name	Data Type	Description
*1 CONSTRAINT_UID	LARGEINT	UID of constraint
*2 CONSTRAINT_TYPE	CHAR(2)	Constraint type: C Check F Foreign P Primary key U Unique
*3 TABLE_UID	LARGEINT	UID of table
4 DISABLED	CHAR(2)	Y if not enforced N if enforced
5 DROPPABLE	CHAR(2)	Y if user can drop N if user cannot drop
6 IS_DEFERRABLE	CHAR(2)	Reserved for future use
7 INITIALLY_DEFERRABLE	CHAR(2)	Reserved for future use
8 INDEX_UID	LARGEINT	If an index supports this constraint, UID of index; otherwise 0 (zero)

Column Name	Data Type	Description
9 ENFORCED	CHAR(2)	Reserved for future use.
10 VALIDATED	CHAR(2)	Reserved for future use.
11 LAST_VALIDATED	LARGEINT	Reserved for future use.

* Indicates primary key

In version 1200 schemas, the primary key consists of the columns in the following order:

1. CONSTRAINT_UID
2. CONSTRAINT_TYPE
3. TABLE_UID

In version 3000 and later schemas, the primary key consists of the columns in the following order:

1. TABLE_UID
2. CONSTRAINT_UID
3. CONSTRAINT_TYPE

TBL_PRIVILEGES Table

TBL_PRIVILEGES is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that stores grant information for tables in the catalog:

Column Name	Data Type	Description
*1 GRANTOR	INT	Security ID of grantor (or of owner if grantor is the super ID acting for owner)
*2 GRANTOR_TYPE	CHAR(2)	S if system grant U if user grant O if granted as schema owner
*3 GRANTEE	INT	If GRANTEE_TYPE is U, security ID of grantee and link to COL_PRIVILEGES; no meaning otherwise
*4 GRANTEE_TYPE	CHAR(2)	P if public grant U if user grant O if grantee is schema owner
*5 TABLE_UID	LARGEINT	UID of table

Column Name	Data Type	Description
*6 PRIVILEGE_TYPE	CHAR(2)	Privilege type: S SELECT I INSERT D DELETE U UPDATE R REFERENCES E EXECUTE (for stored procedures)
7 IS_GRANTABLE	CHAR(2)	Y if granted with grant option N if not

* Indicates primary key

Grant information for individual columns is stored separately in the COL_PRIVILEGES table.

All character columns store letters in uppercase except for GRANTOR and GRANTEE.

In version 1200 schemas, the primary key consists of the columns in the following order:

1. GRANTOR
2. GRANTOR_TYPE
3. GRANTEE
4. GRANTEE_TYPE
5. TABLE_UID
6. PRIVILEGE_TYPE

In version 3000 and later version schemas, the primary key consists of the columns in the following order:

1. TABLE_UID
2. GRANTOR
3. GRANTOR_TYPE
4. GRANTEE
5. GRANTEE_TYPE
6. PRIVILEGE_TYPE

TEXT Table

TEXT is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that stores text for objects in the catalog:

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID for object.
*2 OBJECT_SUB_ID	INT	Value to differentiate between text items associated with the same object
*3 SEQUENCE_NUM	INT	0 if part 1 of text, 1 if part 2 of text, 2 if part 3, and so on
4 TEXT	VARCHAR (3000)	Text associated with object.

* Indicates primary key

The TEXT table stores text for objects in the catalog such as check constraint text, view text, or the Java signature of stored procedures in Java (SPJ). The text is stored in increments of up to 3000 bytes. Text with more than 3000 bytes has multiple entries, ordered as indicated by the SEQUENCE_NUM column.

The format of a compressed Java signature differs when a stored procedure returns result sets because the Java parameters representing result sets do not map with any of the procedure's SQL parameters.

TRIGGERS Table

TRIGGERS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that describes triggers:

Column Name	Data Type	Description
*1 TRIGGER_UID	LARGEINT	UID of trigger object
2 SUBJECT_CATALOG_UID	LARGEINT	UID of the catalog of the subject table
3 SUBJECT_SCHEMA_UID	LARGEINT	UID of the schema of the subject table
4 SUBJECT_UID	LARGEINT	UID of the table on which the trigger is defined
5 ACTIVATION_TIME	CHAR(2)	Activation time: B Before A After
6 OPERATION	CHAR(2)	Operation that fires the trigger: I INSERT D DELETE U UPDATE
7 GRANULARITY	CHAR(2)	Granularity: R Row S Statement

Column Name	Data Type	Description
8 COLUMNS_IMPLICIT	CHAR(2)	Relevant only for UPDATE trigger: Y Yes N No
9 ENABLED	CHAR(2)	Current status of trigger: Y if enabled N if not
10 TRIGGER_CREATED	LARGEINT	Timestamp of creation of the trigger

* Indicates primary key

TRIGGERS_CAT_USAGE Table

TRIGGERS_CAT_USAGE is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that describes a trigger's use of objects in other catalogs (primarily needed for the DROP TRIGGER statement). Triggers can access objects in different ("foreign") catalogs than the catalog of the trigger itself.

Column Name	Data Type	Description
*1 TRIGGER_UID	LARGEINT	UID of trigger object
2 OTHER_CATALOG_UID	LARGEINT	UID of the foreign catalog containing schemas containing objects used by this trigger
*3 OTHER_SCHEMA_UID	LARGEINT	UID of the foreign schema containing objects used by this trigger

* Indicates primary key

The primary key consists of the columns in the following order:

1. TRIGGER_UID
2. OTHER_SCHEMA_UID

TRIGGER_USED Table

TRIGGER_USED is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that contains Information on how triggers use objects. It serves three purposes:

- Given a table, return all the triggers of a certain operation that are defined on that table. Those triggers might be in other catalogs.
- For every local object (table or view), check if a trigger anywhere is using it. This query also describes how the trigger is using the object (for example, SELECT), and might be needed to determine if and how a table has been used or modified.
- For an UPDATE trigger on explicit columns in the subject table, only the TRIGGER_USED table keeps the list of those columns (a row for each column).

Column Name	Data Type	Description
1 TRIGGER_CATALOG_UID	LARGEINT	UID of trigger's catalog
2 TRIGGER_SCHEMA_UID	LARGEINT	UID of trigger's schema
*3 TRIGGER_UID	LARGEINT	UID of trigger object
*4 USED_OBJECT_UID	LARGEINT	UID of the local object used by the trigger
*5 USED_COL_NUM	INT	The column number in USED_OBJECT_UID. When there is no specific column, the value is 1.

Column Name	Data Type	Description
*6 OPERATION	CHAR(2)	Operation that fires the trigger U UPDATE I INSERT D DELETE For the used-object only, the operation performed on the used object S SELECT R ROUTINE
*7 IS_SUBJECT_TABLE	CHAR(2)	Y if the USED_OBJECT_UID is the subject table of this trigger N if the USED_OBJECT_UID is used only by this trigger

* Indicates primary key

The primary key consists of the columns in the following order:

1. USED_OBJECT_UID
2. USED_COL_NUM
3. OPERATION
4. IS_SUBJECT_TABLE
5. TRIGGER_UID

VWS Table

VWS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that lists views in the catalog:

Column Name	Data Type	Description
*1 OBJECT_UID	LARGEINT	UID of view
2 CHECK_OPTION	CHAR(2)	C if CASCADE L if LOCAL N if None
3 IS_UPDATABLE	CHAR(2)	Y if updating allowed N if not
4 IS_INSERTABLE	CHAR(2)	Y if inserting allowed N if not
5 SIMILARITY_CHECK	CHAR(2)	Y if Similarity Check is enabled N if not

* Indicates primary key

Text for views is stored separately in the TEXT table.

Location for views is stored separately in the REPLICAS table.

VW_COL_TBL_COLS Table

VW_COL_TBL_COLS is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that records the base table columns referenced by each column of a view in the catalog:

Column Name	Data Type	Description
*1 VIEW_UID	LARGEINT	UID of referencing view
*2 VIEW_COL_NUM	INT	Column number of referencing column (first is 0)
*3 UNDERLYING_CAT_UID	LARGEINT	UID of catalog of referenced table
*4 UNDERLYING_SCH_UID	LARGEINT	UID of schema of referenced table
*5 UNDERLYING_OBJ_UID	LARGEINT	UID of referenced table
*6 UNDERLYING_COL_NUM	INT	Column number of referenced column (first is 0)

* Indicates primary key

VW_COL_USAGE Table

VW_COL_USAGE is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that records references in views in the catalog to columns of tables or views:

Column Name	Data Type	Description
*1 USING_VIEW_UID	LARGEINT	UID of referencing view
*2 USED_CAT_UID	LARGEINT	UID of catalog of referenced column
*3 USED_SCH_UID	LARGEINT	UID of schema of referenced column
*4 USED_OBJ_UID	LARGEINT	UID of object with referenced column
*5 COLUMN_NUMBER	INT	Column number within object (first is 0)

* Indicates primary key

VW_TBL_USAGE Table

VW_TBL_USAGE is a metadata table in DEFINITION_SCHEMA_VERSION_*vernum* that records references by views in the catalog to tables or other views:

Column Name	Data Type	Description
*1 USING_VIEW_UID	LARGEINT	UID of referencing view
*2 USED_OBJ_UID	LARGEINT	UID of referenced object
*3 VIEW_CATALOG_UID	LARGEINT	UID of catalog of referencing view
*4 USED_OBJ_CATALOG_UID	LARGEINT	UID of catalog of referenced object
5 VIEW_SCHEMA_UID	LARGEINT	UID of schema of referencing view
6 USED_OBJ_SCHEMA_UID	LARGEINT	UID of schema of referenced object

* Indicates primary key

System Defaults Table

SYSTEM_DEFAULTS is a metadata table in the SYSTEM_DEFAULTS_SCHEMA of catalog NONSTOP_SQLMX_*nodename*. that you use to store system-level default settings that override some of the system-defined default settings. NonStop SQL/MX uses system-defined default settings for attributes that are associated with compiling and executing queries. The system-defined default settings, which are hard-coded settings, are optimal under most circumstances. However, in some circumstances, you might want to override specific system-defined default settings.

To update the SYSTEM_DEFAULTS table, you must be the super ID or a user to whom the super ID has granted UPDATE privileges. All other users have SELECT privileges on this table.

SYSTEM_DEFAULTS Table

The SYSTEM_DEFAULTS table remains empty until you insert rows that contain default settings. This table shows the columns of the SYSTEM_DEFAULTS table:

Column Name	Data Type	Description
*1 SUBSYSTEM	VARCHAR(30)	Subsystem name, default SQLMX. This value must be SQLMX because only this subsystem is supported in NonStop SQL/MX.
*2 ATTRIBUTE	VARCHAR(100)	Attribute name.
3 ATTR_VALUE	VARCHAR(1000)	Attribute value.
4 ATTR_COMMENT	VARCHAR(1000)	Comment.

* Indicates primary key

Overriding System-Defined Default Settings

The values that you insert into the SYSTEM_DEFAULTS table override the system-defined default settings. The default settings in the SYSTEM_DEFAULTS table are considered to be system-level default settings because they persist for all sessions that use that SYSTEM_DEFAULTS table.

You can override a system-defined default setting or a default setting in the SYSTEM_DEFAULTS table for the current process, or session, by issuing a CONTROL QUERY DEFAULT statement or a CONTROL TABLE statement.

For some attributes, you can override a default setting by specifying an option within a statement or command. For example, you can set the transaction isolation level with the SELECT statement. If you do not enter an option when entering a command or do not provide some attribute associated with the execution of queries, NonStop SQL/MX uses a default setting.

This table shows the lowest (1) to highest (6) order of precedence for different methods of specifying default settings:

Method of Specifying Default Settings	Scope of the Setting	When Applied
1. System-defined default settings (hard-coded)	System-wide	Installation
2. Default settings in SYSTEM_DEFAULTS table	System-wide	Compile time or by reentering the MXCI session
3. CONTROL QUERY DEFAULT statement	Current process	Compile time or immediately in the MXCI session
4. CONTROL TABLE statement	Current process	Compile time or immediately in the MXCI session
5. SQL statement option	Current process	Compile time or immediately in the MXCI session
6. SET TABLE TIMEOUT statement	Current process	Run time

Inserting Values Into the SYSTEM_DEFAULTS Table

The insertions do not affect your current session. You must exit and then reenter the MXCI session for these values to take effect as system-level default settings. In addition, the insertions do not affect previously compiled modules. You must recompile these modules for the values to take effect.

Changes you make through the SYSTEM_DEFAULTS table are permanent until changed by another UPDATE statement or overridden by a CONTROL QUERY DEFAULT, CONTROL TABLE, SQL statement, or SET TABLE TIMEOUT statement, as noted in the previous table.

Using the CONTROL QUERY DEFAULT Statement

Execution of the CONTROL QUERY DEFAULT statement does not change the contents of the SYSTEM_DEFAULTS table and, therefore, affects only the current session. See CONTROL QUERY DEFAULT Statement on page 2-60.

If an attribute has a value in the SYSTEM_DEFAULTS table and a CONTROL QUERY DEFAULT statement is issued for that attribute, the value specified by CONTROL QUERY DEFAULT takes precedence over the value in the SYSTEM_DEFAULTS table for the current process.

Default Attributes

Default attributes control these activities:

Character Set
Constraint Droppable Options
Data Types
Function Control
Histograms
Isolation Level
Locking
Local Autonomy
Metadata Management
Module Management
Nonaudited Tables
Object Naming
Partition Management
Query Optimization and Performance
Query Plan Caching
Referential Action
Row Maintenance
Scratch Disk Management
Sequence Functions
Statement Atomicity
Statement Recompilation
Stored Procedures in Java
Stream Access
Table Management
Trigger Management

This table provides a quick reference to the attributes you can set or override with the SYSTEM_DEFAULTS table:

Attribute (page 1 of 9)	**Description**	**Category**
ALLOW_DP2_ROW_SAMPLING	Determines whether the sampling is done by the DP2 or the SQL/MX Executor.	Query Optimization and Performance on page 10-66
ANSI_STRING_FUNCTIONALITY	Determines the behavior of the LPAD and RPAD functions. A value set to ON will pad the string with the specified characters. A value set to OFF will replace the string with the specified characters. For more information, see Examples of LPAD and Examples of RPAD.	Function Control on page 10-52
ATTEMPT_ASYNCHRONOUS_ ACCESS	Controls no-wait access for partitions.	Query Optimization and Performance on page 10-66
ATTEMPT_ESP_PARALLELISM	Controls whether the optimizer generates and costs plans that use ESP parallelism.	Query Optimization and Performance on page 10-66
AUTOMATIC_ RECOMPILATION	Determines whether a statement is recompiled if its access plan is no longer valid at run time.	Statement Recompilation on page 10-78
CACHE_HISTOGRAMS	Controls whether the optimizer caches histograms.	Histograms on page 10-52
CACHE_HISTOGRAMS_ REFRESH_INTERVAL	Controls interval at which histograms are refreshed.	Histograms on page 10-52
CATALOG	Default ANSI catalog name.	Object Naming on page 10-60
CHECK_CONSTRAINT_PRUNING	Controls the check constraints pruning optimization.	Query Optimization and Performance on page 10-66
CREATE_DEFINITION_SCHEMA_V ERSION	Assigns schema version to new schemas.	Metadata Management on page 10-59
CROSS_PRODUCT_CONTROL	Determines whether plans are eliminated that contain unnecessary cross-products.	Query Optimization and Performance on page 10-66

Attribute (page 2 of 9)	**Description**	**Category**
DATA_FLOW_OPTIMIZATION	Controls whether query plans are considered that have high data flow rates.	Query Optimization and Performance on page 10-66
DDL_DEFAULT_ LOCATIONS	Specifies the physical location of the primary partition to be created by CREATE statements that do not specify a LOCATION clause.	Partition Management on page 10-63
DDL_VIEW_SIMILARITY_CHECK	Specifies whether views by default are created with Similarity Check enabled or disabled.	Query Optimization and Performance on page 10-66
DEF_MAX_HISTORY_ ROWS	Default for number of rows the SEQUENCE BY operator keeps in its history buffer.	Sequence Functions on page 10-77
DEFAULT_BLOCKSIZE on page 10-81	enables you to change the default behavior when database objects are created that do not specify a BLOCKSIZE.	Table Management on page 10-81
DOOM_USERTRANSACTION	Controls whether NonStop SQL/MX dooms a transaction when it encounters an unrecoverable error.	Statement Atomicity on page 10-78
DP2_CACHE_4096_BLOCKS	Specifies blocks allocated for disk cache.	Query Optimization and Performance on page 10-66
DYNAMIC_HISTOGRAM_ COMPRESSION	Reduces the number of histogram intervals for histograms of base table columns when those histograms are read from disk.	Histograms on page 10-52
FFDC_DIALOUTS_FOR_MXCMP	Controls whether FFDC dial-outs should occur when the compiler terminates abnormally or detects an internal error.	Query Optimization and Performance on page 10-66
FLOATTYPE	Controls whether the output of FLOAT data types should be treated as Tandem FLOAT or IEEE FLOAT.	Data Types on page 10-51

Attribute (page 3 of 9)	**Description**	**Category**
GENERATE_EXPLAIN	Controls whether EXPLAIN output is generated.	Query Optimization and Performance on page 10-66
GEN_EIDR_BUFFER_SIZE	Buffer size for partition access.	Query Optimization and Performance on page 10-66
GEN_MAX_NUM_PART_DISK_ENTRIES	Controls the size of a partition list prepared by the compiler and used by the executor.	Partition Management on page 10-63
GEN_MAX_NUM_PART_NODE_ENTRIES	Controls the size of a partition list prepared by the compiler and used by the executor.	Partition Management on page 10-63
GEN_PA_BUFFER_SIZE	Buffer size for partition access.	Query Optimization and Performance on page 10-66
HIST_DEFAULT_SEL_FOR_LIKE_WILDCARD	Specifies the selectivity factor used by the optimizer for LIKE predicates where the matched pattern starts with a wildcard.	Histograms on page 10-52
HIST_DEFAULT_SEL_FOR_PRED_RANGE	Specifies the selectivity factor used by the optimizer for range predicates when current histogram statistics do not exist.	Histograms on page 10-52
HIST_JOIN_CARD_LOWBOUND	Controls join cardinality.	Histograms on page 10-52
HIST_NO_STATS_REFRESH_INTERVAL	Controls the interval at which default statistics are refreshed.	Histograms on page 10-52
HIST_NO_STATS_ROWCOUNT	Estimated row count when histogram statistics do not exist.	Histograms on page 10-52
HIST_NO_STATS_UEC	Estimated unique entry count (UEC) when histogram statistics do not exist.	Histograms on page 10-52
HIST_PREFETCH	Determines if histograms are prefetched for caching.	Histograms on page 10-52

Attribute (page 4 of 9)	Description	Category
HIST_ROWCOUNT_ REQUIRING_STATS	Minimum row count that determines when warnings are issued to update statistics.	Histograms on page 10-52
HIST_SAME_TABLE_PRED_ REDUCTION	Controls overlap amount in predicate selectivity when multicolumn predicates are used.	Histograms on page 10-52
HIST_SCRATCH_VOL	Sets the physical volume for UPDATE STATISTICS temporary tables.	Histograms on page 10-52
HIST_SECURITY_WARNINGS	Controls whether warnings on histogram tables are displayed.	Histograms on page 10-52
INDEX_ELIMINATION_LEVEL	Indicates the degree of heuristic elimination of indexes consideration by the optimizer.	Query Optimization and Performance on page 10-66
INFER_CHARSET	Enable character set inference for ODBC 2.X.	Character Set on page 10-49
INSERT_VSBB	Controls method of inserting rows into a table.	Row Maintenance on page 10-75
INTERACTIVE_ACCESS	Determines whether the compiler selects index-based access plans.	Statement Recompilation on page 10-78
ISOLATION_LEVEL	Default transaction isolation level.	Isolation Level on page 10-56
IUD_NONAUDITED_ INDEX_MAINT	Controls whether NonStop SQL/MX allows insert/ update/delete operations on nonaudited tables that require index maintenance.	Nonaudited Tables on page 10-60
JOIN_ORDER_BY_USER	Enables or disables join order you specify in the FROM clause of a query.	Query Optimization and Performance on page 10-66
MATERIALIZE	Default for whether inner tables of join operations with streams are materialized.	Stream Access on page 10-80
MAX_ESPS_PER_CPU_PER_OP	Maximum number of ESPs the optimizer considers starting for each CPU for a given operator.	Query Optimization and Performance on page 10-66

Attribute (page 5 of 9)	**Description**	**Category**
MAX_ROWS_LOCKED_ FOR_STABLE_ACCESS	Maximum number of rows that are locked in STABLE ACCESS mode.	Locking on page 10-57
MDAM_SCAN_METHOD	Enables or disables the MultiDimensional Access Method.	Query Optimization and Performance on page 10-66
MEMORY_USAGE_SAFETY_NET	Specifies the MXCMP memory threshold in megabyte.	Query Optimization and Performance on page 10-66
MIN_MAX_OPTIMIZATION	Enables or disables MIN-MAX optimization.	Query Optimization and Performance on page 10-66
MP_SUBVOLUME	Default NonStop operating system Guardian subvolume.	Object Naming on page 10-60
MP_SYSTEM	Default NonStop operating system Guardian system name.	Object Naming on page 10-60
MP_VOLUME	Default NonStop operating system Guardian volume.	Object Naming on page 10-60
MSCF_ET_REMOTE_MSG_ TRANSFER	Factors in the cost of transferring messages to and from a remote node.	Query Optimization and Performance on page 10-66
MULTIUNION	Controls the MultiUnion operator.	Query Optimization and Performance on page 10-66
MXCMP_PLACES_LOCAL_ MODULES	Determines where globally placed modules are generated.	Module Management on page 10-59
NAMETYPE	Default for the use of three-part logical names (ANSI) or four-part Guardian names (NSK).	Object Naming on page 10-60
NATIONAL_CHARSET	Default character set for the use of NCHAR.	Character Set on page 10-49
NOT_NULL_CONSTRAINT_ DROPPABLE_OPTION	Default for DROPPABLE (ON) or NOT DROPPABLE for NOT NULL constraint.	Constraint Droppable Options on page 10-50
NUMBER_OF_USERS	Number of users that can run concurrent queries that use large amounts of memory.	Query Optimization and Performance on page 10-66

Attribute (page 6 of 9)	**Description**	**Category**
OLT_QUERY_OPT	Enables a fast path evaluation method for certain simple SQL queries.	Query Optimization and Performance on page 10-66
OPTIMIZATION_LEVEL	Controls increasing effort in optimizing queries.	Query Optimization and Performance on page 10-66
OPTS_PUSH_DOWN_DAM	Controls whether NonStop SQL/MX considers push-down plans.	Query Optimization and Performance on page 10-66
PARALLEL_NUM_ESPS	Maximum number of parallel ESPs that work on a particular type of operator, like a join.	Query Optimization and Performance on page 10-66
PM_OFFLINE_ TRANSACTION_ GRANULARITY	Number of rows to be copied in an offline MODIFY transaction.	Partition Management on page 10-63
PM_ONLINE_ TRANSACTION_ GRANULARITY	Number of rows to be copied in an online MODIFY transaction.	Partition Management on page 10-63
POS_LOCATIONS	Controls location of partitions to be automatically created.	Partition Management on page 10-63
POS_NUM_OF_PARTNS	Controls number of partitions to be automatically created.	Partition Management on page 10-63
POS_RAISE_ERROR	Determines whether error should be displayed.	Partition Management on page 10-63
PREFERRED_PROBING_ ORDER_FOR_NESTED_JOIN	Controls whether rows of the inner table must be read in key order of the access path.	Query Optimization and Performance on page 10-66
PRESERVE_MIN_SCALE	Allows you to preserve minimum scale in a result when the precision exceeds 18.	Function Control on page 10-52
PRIMARY_KEY_CONSTRAINT_ DROPPABLE_OPTION	Default for DROPPABLE (ON) or NOT DROPPABLE for PRIMARY KEY constraint.	Constraint Droppable Options on page 10-50

Attribute (page 7 of 9)	**Description**	**Category**
QUERY_CACHE	Controls the maximum amount of memory that the SQL/MX compiler is allowed to use for holding the cached plans of previously compiled queries.	Query Plan Caching on page 10-73
QUERY_CACHE_MAX_VICTIMS	Limits the number of entries that an unusually large query plan is allowed to displace from the cache.	Query Plan Caching on page 10-73
QUERY_CACHE_REQUIRED_ PREFIX_KEYS	Determines how many and which columns of a composite primary or partition key are required for an equality key predicate to be considered cacheable.	Query Plan Caching on page 10-73
QUERY_CACHE_STATEMENT_ PINNING	Controls the pinning and unpinning of query cache entries.	Query Plan Caching on page 10-73
READONLY_CURSOR	Controls whether FOR UPDATE is required for cursor declarations for columns to be updatable.	Row Maintenance on page 10-75
RECOMPILE_ON_ PLANVERSION_ERROR	Determines whether a statement is recompiled if its access plan is no longer valid at run time due to versioning errors.	Statement Recompilation on page 10-78
RECOMPILATION_ WARNINGS	Determines whether a warning is returned when a statement is dynamically recompiled.	Statement Recompilation on page 10-78
REF_CONSTRAINT_NO_ ACTION_LIKE_RESTRICT	Determines how NonStop SQL/MX handles referential action in ALTER TABLE and CREATE TABLE statements.	Referential Action on page 10-75
REMOTE_ESP_ALLOCATION	Identifies the scope the optimizer considers when determining the active systems.	Query Optimization and Performance on page 10-66

Attribute (page 8 of 9)	**Description**	**Category**
SAVE_DROPPED_TABLE_DDL	Controls whether definitions of dropped tables or partitions are saved, to enable them to be recovered.	Table Management on page 10-81
SCHEMA	Default ANSI schema name.	Object Naming on page 10-60
SCRATCH_DISKS	Restricts scratch disks for sort operations to the volumes specified.	Scratch Disk Management on page 10-76
SCRATCH_DISKS_EXCLUDED	Excludes certain volumes from being used as scratch disks for sort operations.	Scratch Disk Management on page 10-76
SCRATCH_DISKS_PREFERRED	Volumes preferred as scratch disks for sort operations.	Scratch Disk Management on page 10-76
SCRATCH_FREESPACE_THRESHOLD_PERCENT	Amount of scratch space left on disks as a threshold.	Scratch Disk Management on page 10-76
SIMILARITY_CHECK	Determines whether similarity checks are made to either keep or recompile an access plan.	Statement Recompilation on page 10-78
SKIP_UNAVAILABLE_PARTITION	controls whether SQL continues to process a query when a partition required by the access plan of the query is unavailable.	Local Autonomy on page 10-58
SORT_MAX_HEAP_SIZE_MB	Allocates a default value to the heap memory size for operations involving the sort operator.	Query Optimization and Performance on page 10-66
STREAM_TIMEOUT	Default time for a fetch operation using stream access to wait for more rows before timing out.	Stream Access on page 10-80
TABLELOCK	Default when table locks are used.	Locking on page 10-57
TEMPORARY_TABLE_HASH_PARTITIONS	Specifies partitioning for trigger temporary tables.	Trigger Management on page 10-83

Attribute (page 9 of 9)	**Description**	**Category**
TIMEOUT	Default time to wait for a lock before NonStop SQL/MX returns a timeout error.	Locking on page 10-57
UNION_TRANSITIVE_PREDICATES	Controls the union transitive predicates.	Query Optimization and Performance on page 10-66
UDR_JAVA_OPTIONS	Specifies JVM startup options for the Java environment of an SPJ.	Stored Procedures in Java on page 10-80
UPD_ORDERED	Controls whether rows must be inserted, updated, or deleted in clustering key order.	Query Optimization and Performance on page 10-66
UPD_ABORT_ON_ERROR	Controls whether an update, insert, or delete function is aborted if an error occurs.	Statement Atomicity on page 10-78
UPD_SAVEPOINT_ON_ERROR	Controls whether DP2 savepoints should be used and if the transaction should be aborted in case of an error.	Statement Atomicity on page 10-78
VARCHAR_PARAM_DEFAULT_SIZE	Controls the allowable length of an untyped parameter, which is typed as VARCHAR during the compilation of a query.	Table Management on page 10-81
ZIG_ZAG_TREES	Enables or disables the optimizer to consider zig-zag trees in addition to linear trees.	Query Optimization and Performance on page 10-66

For more information, see the description for the individual attribute.

Character Set

This attribute determines the default for the character set:

Attribute	Setting

INFER_CHARSET	When set to ON, the parser does not consider the character set of literals. However, the binder decides the character set depending on the context. The default value for this attribute is OFF for MXCI and ON for ODBC/MX and JDBC/MX.
	For example:
	`>>create table infchar(i CHAR(10) character set ucs2);`
	`--- SQL operation complete.`
	`>>insert into infchar values('abc');`
	`*** ERROR[4039] Column I is of type CHAR(10) CHARACTER SET UCS2, incompatible with the value's type, CHAR(3) CHARACTER SET ISO88591.`

	`*** ERROR[8822] The statement was not prepared.`
	`>>control query default infer_charset 'on';`
	`--- SQL operation complete.`
	`>>insert into infchar values('abc');`
	`--- 1 row(s) inserted.`

NATIONAL_CHARSET	Displays the national character set ISO88591, UCS2, KANJI, or KSC5601 used in NCHAR and NCHAR VARYING columns. The national character set also governs the interpretation of the character string literal N'*string*'. You select the national character set when you install NonStop SQL/MX by using the `-n` option of the `InstallSqlmx` script. If you specify KANJI or KSC5601 and later attempt to create an SQL/MX table with an NCHAR column, you will receive an error message because SQL/MX tables do not support the KANJI or KSC5601 character sets. If you do not specify a value for the `-n` option, the national character set defaults to UCS2. For more information about setting the national character set from the `InstallSqlmx` script, see the *SQL/MX Installation and Management Guide*. For more information about the use of the N'string' literal, see Character String Literals on page 6-64. For more information about the use of the NCHAR keyword, see Character String Data Types on page 6-22.

Constraint Droppable Options

These table entries describe settings that enable NonStop SQL/MX to ensure that the defaults for certain constraints are set to NOT DROPPABLE:

Attribute	Setting
NOT_NULL_CONSTRAINT_ DROPPABLE_OPTION	Set to ON (DROPPABLE) or OFF (NOT DROPPABLE). This option is used if DROPPABLE or NOT DROPPABLE does not appear in the definition of a NOT NULL column constraint. The default is OFF.
PRIMARY_KEY_CONSTRAINT_ DROPPABLE_OPTION	Set to ON (DROPPABLE) or OFF (NOT DROPPABLE). This option is used if DROPPABLE or NOT DROPPABLE does not appear in the definition of a PRIMARY KEY constraint. The default is OFF.

These settings affect the way NonStop SQL/MX processes NOT NULL and PRIMARY KEY constraints, as follows:

- If a column is defined with the NOT NULL NOT DROPPABLE constraint, the executor does not check for null—thereby improving performance of updates and inserts. The NOT NULL NOT DROPPABLE constraint also eliminates the need for a null indicator, which reduces space requirements.
- If a column or a column list within a table is defined with the PRIMARY KEY NOT DROPPABLE constraint, the primary key column or column list can be used as a storage key—the most efficient method for partitioning by values of a unique key. In this case, a separate index is not required for the primary key.

Data Types

This attribute controls whether the output of dynamic SELECT statements and dynamic parameters that are FLOAT data types should be treated as Tandem FLOAT format or IEEE FLOAT format:

Attribute	Setting
FLOATTYPE	Set to IEEE or TANDEM. The default is TANDEM.

Function Control

This attribute controls how NonStop SQL/MX handles functions:

Attribute	Setting
PRESERVE_MIN_SCALE	An arithmetic operation on numeric columns might give a wrong result. If the result value exceeds the allowed numeric limit, the precision is truncated to 18. PRESERVE_MIN_SCALE allows you to preserve minimum scale in a result when the precision exceeds 18. Allowable values: 0 to 18 The default is 0.
ANSI_STRING_FUNCTIONALITY	This CQD determines whether the behavior of SQL string functions is in accordance with the ANSI standards. When set to ON, the SQL string functionality is in accordance with ANSI standards. When set to OFF, the SQL string functionality might not be according to ANSI standards. This CQD is applicable only for the LPAD and RPAD string functions. For more information about these functions, see LPAD Function on page 8-99 and RPAD Function on page 8-150. The default is OFF.

Histograms

These attributes enable NonStop SQL/MX to improve performance of query execution by ensuring defaults for histogram statistics:

Attribute	Setting
CACHE_HISTOGRAMS	Set to ON or OFF. When set to ON, NonStop SQL/MX caches the histogram so that it can be retrieved from the cache rather than from the disk for future queries on the same table. Histogram caching provides faster access to histograms. This attribute significantly reduces compile time for less complex queries. If OFF, histograms cached previously are flushed from cache, and histograms for every query are loaded from the disk. When CACHE_HISTOGRAMS is turned ON again, histograms are reloaded, and NonStop SQL/MX caches them again. The attribute HIST_PREFETCH on page 10-54 also relates to histogram caching. The default is ON.

Attribute	Setting
CACHE_HISTOGRAMS_ REFRESH_INTERVAL	Controls the time interval, in seconds, at which histograms in the histogram cache are refreshed. This is the maximum time a histogram in the cache can be out of date. Allowable values: 0 through 4294967295. The default value is 3600 seconds.
DYNAMIC_HISTOGRAM_ COMPRESSION	Set to ON or OFF. When set to ON, NonStop SQL/MX reduces compile time by reducing the number of histogram intervals. Histogram interval reduction has more affect on complex queries, especially if the underlying data distribution is evenly distributed. The compiler reduces the number of histogram intervals for columns containing numeric data types and nonnumeric data type columns only if there is no join or range predicate. The default is ON.
HIST_DEFAULT_SEL_FOR_ LIKE_WILDCARD	Specifies the selectivity factor used by the optimizer for LIKE predicates where the matched pattern starts with a wildcard, for example, the predicate (user_email LIKE '%.net'). The value is expressed as a percentage so that .10 is 10 percent and .333 is 33.3 percent. Allowable values: 0 through 1. The default value is 0.10.
HIST_DEFAULT_SEL_FOR_ PRED_RANGE	Specifies the selectivity factor used by the optimizer for range predicates when current histogram statistics do not exist. This default is also used for the selectivity of range predicates involving host variables or parameters, for example, the predicate (quantity > ?p1). The value is expressed as a percentage so that .333 is 30 percent. Allowable values: 0 through 1. The default value is 0.333.
HIST_JOIN_CARD_ LOWBOUND	NonStop SQL/MX uses certain assumptions about the relationship between columns from different tables that are involved in a join. In case of insufficient multicolumn statistics, these assumptions might result in underestimating the join cardinality result. The estimated cardinality of the join should not be less than a percentage of the cardinality of the smallest table involved in the join. This default specifies this percentage or fraction value. Allowable values: 0 through 1. The default value of 1.0 corresponds to a join cardinality lower bound equal to 100 percent of the cardinality of the smallest table in the join. A value of 0 means that there is no lower bound limit applied to the join cardinality.

Attribute	Setting
HIST_NO_STATS_REFRESH_ INTERVAL	Specifies the time interval, in seconds, at which default statistics are refreshed. Default statistics are compiler generated statistics for tables for which no UPDATE STATISTICS has been performed. You can change the value of HIST_NO_STATS_REFRESH_INTERVAL if you do frequent inserts and deletes on such tables, for example, a temporary table. If you set this value to 0, default statistics are never cached, and new default statistics are generated by the compiler for every statement, based on the current tables' sizes. Allowable values: 0 through 4294967295. The default value is 3600 seconds.
HIST_NO_STATS_ROWCOUNT	Estimated row count when current histogram statistics do not exist for a table. Used with HIST_NO_STATS_UEC. Adjust these settings when the query execution plan shows that costing values are incorrect because of a lack of statistics on a table involved in the query. Allowable values: 1 through 3.402823466E+38. The default is 100 rows.
HIST_NO_STATS_UEC	Estimated unique entry count (UEC) when current histogram statistics do not exist. Used with HIST_NO_STATS_ROWCOUNT. HIST_NO_STATS_UEC must be less than or equal to HIST_NO_STATS_ROWCOUNT. Allowable values: 1 through 3.402823466E+38. The default value is 2.
HIST_PREFETCH	Set to ON or OFF. When set to ON, NonStop SQL/MX determines if histograms are prefetched for caching. The compiler fetches histograms for all the columns of a table and places them in the cache to improve optimizer performance. The CACHE_HISTOGRAMS attribute must be set to ON for histograms to be prefetched. If OFF, histograms are cached only for columns of a table that is involved in a statement. The default setting is ON.

Attribute	Setting
HIST_ROWCOUNT_ REQUIRING_STATS	Row count that determines when SQLCODE 6007/6008 warnings are issued, which mean statistics have not been updated for all table columns in a query. Only columns from tables that have more rows than this count force these warnings. To avoid these warnings, set this value to a very large number, provided no tables exist with that number of rows in the database. If a file is fragmented, NonStop SQL/MX cannot estimate an accurate row count, and you can receive warning 6008 even if you have set this value to a very large number. You should execute UPDATE STATISTICS to make its histogram information current. The default is 50000 rows. Allowable values: 1 to 3.402823466E+38.
HIST_SAME_TABLE_PRED_ REDUCTION	Controls the amount of overlap in predicate selectivity. Set to a value between 0 (no overlap) and 1 (complete overlap). Affects plans that have multiple predicates on the same table, where multicolumn statistics are not available for the columns in the predicates. The default is 0.0. Allowable values: 0 to 1.
HIST_SCRATCH_VOL	Sets the physical volumes used tor UPDATE STATISTICS temporary files, specified as `'$volume'` with the volume name or names enclosed in single quotes.You may specify multiple locations separated by commas. Every volume specified in the list must be unique. NonStop SQL/MX will calculate how many partitions are needed based on the sample set retrieved by the SAMPLE option. If NonStop SQL/MX determines that it needs more disks than you specified in that option, it will use all the disks you list for this attribute. You may create as many partitions as there are CPUs on the local node. You should distribute the partitions evenly across the CPUs on the local node. That is, specify volumes so that the first volume in the list is controlled by CPU0, the second volume is controlled by CPU1, the third volume is controlled by CPU2, the fourth volume is controlled by CPU3, and so on. Only SQL/MX temporary tables may be hash partitioned. If you specify more than one volume for a SQL/MP temporary table, only the first volume will be used. The default is a blank. If you do not set this value, NonStop SQL/MX uses the default volume specified by the _DEFAULTS define for SQL/MX tables, and the volume of the table’s primary partition for SQL/MP tables.

Attribute	Setting
HIST_SECURITY_WARNINGS	Controls whether MXCMP displays a warning if the user does not have access permissions to statistics tables and the user table's estimated rowcount is greater than the HIST_ROWCOUNT_REQUIRING_STATS value. If set to ON, the compiler reports this warning. If set to OFF, the compiler does not report a warning. The default is ON.

For more information about histogram statistics, see the *SQL/MX Query Guide*.

Isolation Level

This attribute determines how NonStop SQL/MX assigns transaction isolation levels:

Attribute	Setting
ISOLATION_LEVEL	The isolation level for a transaction. Set to: READ UNCOMMITTED READ COMMITTED REPEATABLE READ SERIALIZABLE The default is READ COMMITTED. See Transaction Isolation Levels on page 1-23.

Transaction isolation levels are determined according to rules applied in this order:

1. If you specify an access option explicitly in a DML statement, the SQL/MX compiler compiles the statement with the access option. This access option overrides the isolation level of any containing transactions.
2. If there are no individual statement access options and you issue a SET TRANSACTION ISOLATION LEVEL statement, the SQL/MX compiler uses the setting determined by this SET TRANSACTION statement as the isolation level for the next transaction. See SET TRANSACTION Statement on page 2-376.
3. If you do not specify a SET TRANSACTION statement and you issue a CONTROL QUERY DEFAULT ISOLATION_LEVEL statement, the CONTROL QUERY DEFAULT statement determines the isolation level.
4. If you do not issue a CONTROL QUERY DEFAULT ISOLATION_LEVEL statement, NonStop SQL/MX uses the ISOLATION_LEVEL setting in the SYSTEM_DEFAULTS table if it exists.
5. If you do not specify isolation-level settings, NonStop SQL/MX uses the system-defined isolation level, which is READ COMMITTED.

Locking

These attributes determine how NonStop SQL/MX locks objects:

Attribute	Setting
MAX_ROWS_LOCKED_ FOR_STABLE_ACCESS	The maximum number of rows locked by DP2 in STABLE ACCESS mode before the buffer is returned to the file system. The number of rows which are actually locked depends on this number and the size of the buffer. To increase concurrency, you can decrease this value so that more messages are used to return the same amount of data. The default value is 1.
TABLELOCK	Set to SYSTEM, ON, or OFF to indicate whether the system determines when table locks are to be used for accessing the table or view (SYSTEM), table locks are always used (ON), or table locks are not used (OFF). The default is SYSTEM. For more information on table locks, see Database Integrity and Locking on page 1-11.
TIMEOUT	The time in hundredths of seconds to wait for a lock before returning an error. The range of values you can enter is from -1 to 2147483647. The value -1 directs NonStop SQL/MX not to time out. The value 0 directs NonStop SQL/MX not to wait for a table lock. If the lock cannot be acquired, an error is returned immediately. The default is 6000 in hundredths of seconds, which is equivalent to 60 seconds. This default is valid for compile-time timeout. For run-time timeout, see the SET TABLE TIMEOUT Statement on page 2-372.

If you issue a CONTROL TABLE statement for the TABLELOCK or TIMEOUT option, the specified control table value overrides the system-defined default setting. See CONTROL TABLE Statement on page 2-74.

Local Autonomy

This attribute controls how SQL/MX handles local autonomy:

Attribute	Setting
SKIP_UNAVAILABLE_PARTITION	This attribute provides local autonomy for certain situations by directing SQL/MX to continue processing a query even if partitions required for the access plan of the query are not available. This attribute applies to partitioned tables, but affects only the main query in a SELECT statement without INTO clause. (SQL always stops processing and returns an error when a required partition is unavailable for a SELECT statement in the search condition of UPDATE or DELETE statement, or any other DDL or DML statement.) If set to ON, SQL/MX continues processing the query even if one or more partitions required for the query plan are not available. A warning message is displayed for each partition that is unavailable. For certain simple SQL queries, such as, single table unique select, SQL/MX enables Online Transaction (OLT) optimization. In such cases, this CQD has no impact. For more information about OLT optimization, see the *SQL/MX query guide*. If set to OFF, an error is displayed indicating that partition is not available. The default setting is OFF.

Metadata Management

This attribute enables NonStop SQL/MX to manage metadata:

Attribute	Setting
CREATE_DEFINITION_SCHEMA_VERSION	Assigns a schema version to new schemas during schema creation time. For SQL/MX 3.0, the valid schema versions are SYSTEM, 1200, and 3000. The default value is SYSTEM. The following scenarios explain how the schema version is assigned: ● When CREATE_DEFINITION_SCHEMA_VERSION is set to SYSTEM: ○ if no user schemas exist in the affected catalog, the new schema will use the current schema version ○ if user schemas exist in the affected catalog, the new schema will use the version of the existing schemas. ● When CREATE_DEFINITION_SCHEMA_VERSION is set to any other value: ○ if that value differs from the version of the existing schemas in a catalog at the schema creation time, error 25221 is raised. ○ If that value is not a valid schema version, error 25222 is raised at the schema creation time.

Module Management

This attribute determines where globally placed modules are generated:

Attribute	Setting
MXCMP_PLACES_LOCAL_MODULES	Set to ON or OFF. If OFF, NonStop SQL/MX generates global modules in the USERMODULES directory. If ON and you do not specify `mxcmp -g moduleLocal=`*`OSS-dir`* on the command line, compiled modules are placed in the current OSS directory. If ON and you specify `mxcmp -g moduleLocal=`*`OSS-dir`* on the command line, compiled modules are placed in the *`OSS-dir`* directory. The default is OFF. For more information about module management, see the *SQL/MX Programming Manual for C and COBOL*.

Nonaudited Tables

This attribute enables NonStop SQL/MX to handle inserts, updates, and deletes against nonaudited SQL/MP tables:

△ **Caution.** If the IUD_NONAUDITED_INDEX_MAINT is set to ON, NonStop SQL/MX allows DML operations on nonaudited tables without error or warning. Before you set this attribute, see the *SQL/MX Comparison Guide for SQL/MP Users* to understand index maintenance of nonaudited tables.

Attribute	Setting
IUD_NONAUDITED_INDEX_MAINT	Set to ON, OFF, or WARN. Specifies whether NonStop SQL/MX allows insert/update/delete operations on nonaudited SQL/MP tables that require index maintenance. If OFF, DML operations on nonaudited tables are not allowed when the tables require index maintenance. Any effort to prepare or compile such a statement results in an error. If WARN, NonStop SQL/MX allows the DML operation when the tables require index maintenance; however, a warning is given. If ON, NonStop SQL/MX allows the operations without error or warning. The default is OFF. SQL/MX tables must be audited.

For more information about the differences between NonStop SQL/MP and NonStop SQL/MX relating to DML operations against nonaudited tables, see the *SQL/MX Comparison Guide for SQL/MP Users*.

Object Naming

These attributes determine how NonStop SQL/MX assigns object names:

Attribute	Setting
NAMETYPE	Set to ANSI or NSK to indicate whether the system uses three-part logical names (ANSI) or four-part physical Guardian names (NSK) to refer to database objects in statements. The NSK setting applies to the resolution of unqualified SQL/MP object names. The default is ANSI.
CATALOG	Default catalog name, used if no first part is specified in a three-part logical name. If not set, the group name of the current user becomes the default first part of the logical name. In SQL/MX Release 2.0, the three-part name is an ANSI name. The parts catalog and schema denote the ANSI-defined catalog and schema.

Attribute	Setting
SCHEMA	Default schema (without catalog) name, used if no second part is specified in a three-part logical name. The schema name can be qualified by a catalog name, in which case this catalog name supersedes any settings for the CATALOG attribute. If not set, the user name of the current user becomes the default second part of the logical name. In SQL/MX Release 2.0, the three-part name is an ANSI name. The parts catalog and schema denote the ANSI-defined catalog and schema.
MP_SUBVOLUME	Default physical subvolume, used if no subvolume is specified in the Guardian name. If not set, the default subvolume is specified by the =_DEFAULTS define.
MP_SYSTEM	Default system name, used if no system is specified in the NSK name. If not set, the default system is specified by the =_DEFAULTS define.
MP_VOLUME	Default physical volume, used if no volume is specified in the NSK name. If not set, the default volume is specified by the =_DEFAULTS define.

The three-part logical name of the form `catalog.schema.object` is an ANSI name. The parts `catalog` and `schema` denote the ANSI-defined catalog and schema.

To be compliant with ANSI SQL:1999, NonStop SQL/MX provides support for ANSI three-part object names. By using these names, you can develop ANSI applications that access all SQL/MP objects. You must create an alias for SQL/MP objects. See CREATE SQLMP ALIAS Statement on page 2-104 and ALTER SEQUENCE Statement on page 2-13 for more information.

NAMETYPE Attribute

The NAMETYPE attribute determines the precedence rules for object naming according to whether the value of the attribute is ANSI or NSK. The value of the NAMETYPE attribute is determined according to rules applied in this order:

1. The SET NAMETYPE statement and CONTROL QUERY DEFAULT statement have the same precedence:

   - If you issue a SET NAMETYPE statement, the compiler uses the setting determined by this statement as the value of the attribute.

     Use the SET NAMETYPE statement in MXCI. For embedded SQL, the SET NAMETYPE statement affects only dynamic embedded SQL. The DECLARE NAMETYPE statement affects only static embedded SQL.

   - For SQL statements issued through MXCI and embedded SQL, the compiler uses the NAMETYPE value set by the CONTROL QUERY DEFAULT statement (if issued).

2. For SQL statements issued through MXCI and embedded SQL, the compiler uses the SYSTEM_DEFAULTS table entry (if it exists).
3. For SQL statements issued through MXCI and embedded SQL, if these values are not set, the system-defined default setting for the NAMETYPE attribute is ANSI.

Attribute Value ANSI for Logical Names

If the NAMETYPE attribute is ANSI or is not specified, object names are determined according to rules applied in this order:

1. If you specify a fully qualified three-part logical name explicitly in your SQL statement, the SQL/MX compiler compiles the statement by using the three parts of this name.
2. The SET CATALOG statement, SET SCHEMA statement, and CONTROL QUERY DEFAULT statement have the same precedence:
   - If you do not specify a fully qualified logical name and you issue a SET CATALOG or SET SCHEMA statement, the compiler uses the setting determined by these statements as the current catalog (the first part) or schema (the second part).

     Use the SET CATALOG and SET SCHEMA statements in MXCI. For embedded SQL, the SET CATALOG and SET SCHEMA statements affect only dynamic embedded SQL. The DECLARE CATALOG and DECLARE SCHEMA statements affect only static embedded SQL.
   - For SQL statements issued through MXCI and embedded SQL, the compiler uses the CATALOG or SCHEMA values set by CONTROL QUERY DEFAULT statements (if issued).
3. For SQL statements issued through MXCI and embedded SQL, the compiler uses the CATALOG or SCHEMA values in the SYSTEM_DEFAULTS table (if they exist).
4. For SQL statements issued through MXCI and embedded SQL, if these values are not set, the system-defined default setting for `catalog.schema` is `group.user`, which is the current user ID.

Attribute Value NSK for Guardian Names and Guardian Name Resolution

If the NAMETYPE attribute is set to NSK, object names are determined according to rules applied in this order:

1. If you specify a fully qualified physical name explicitly in your SQL statement, the SQL/MX compiler compiles the statement by using the four parts of this name: the physical system, volume, subvolume, and file names.
2. The SET MPLOC statement and CONTROL QUERY DEFAULT statement have the same precedence:
   - If you do not specify a fully qualified object name and you issue a SET MPLOC statement, the compiler uses the setting determined by these statements as

the current physical location. SET MPLOC changes the values for default attributes MP_SYSTEM, MP_VOLUME and MP_SUBVOLUME.

Use the SET MPLOC statement in MXCI. For embedded SQL, the SET MPLOC statement affects only dynamic embedded SQL. The DECLARE MPLOC statement affects only static embedded SQL.

- For SQL statements issued through MXCI and embedded SQL, the compiler uses the MP_SYSTEM, MP_VOLUME or MP_SUBVOLUME values set by CONTROL QUERY DEFAULT statements (if issued).

3. For SQL statements issued through MXCI and embedded SQL, the compiler uses the MP_SYSTEM, MP_VOLUME or MP_SUBVOLUME values in the SYSTEM_DEFAULTS table (if they exist).
4. For SQL statements issued through MXCI and embedded SQL, if these values are not set, the system-defined default setting for $*volume.subvol* is specified by the =_DEFAULTS define.

   The TACL command INFO DEFINE =_DEFAULTS can be used to look at the default volume and subvolume for the current TACL session. For more information on the =_DEFAULTS define, see the *Guardian Programmer's Guide*.

Partition Management

These attributes are used by NonStop SQL/MX for partition management:

Attribute	Setting
DDL_DEFAULT_ LOCATIONS	Physical location of the primary range partition to be created by CREATE statements that do not specify a LOCATION clause, specified as [*node*.]$*volume.* You can specify multiple locations separated by commas. If you enter a CREATE TABLE or CREATE INDEX or CREATE PROCEDURE statement without specifying a LOCATION clause and you have specified this default, the location is determined by picking one volume from the specified list. If the statement does not have a LOCATION clause and you do not specify this default or set it to space (" "), NonStop SQL/MX uses the value of the =_DEFAULTS environment variable (the default volume) for the partition location. If you enter a CREATE CATALOG statement without specifying a LOCATION clause and you have specified this default, the location of the catalog's metadata is determined by picking one volume from the specified list. The default is no specification. **Note:** This attribute is available only to systems running NonStop SQL/MX Release 2.1 or later.

Attribute	Setting
GEN_MAX_NUM_PART_ DISK_ENTRIES	When a statically compiled statement references partitioned objects, this default is used to control the size of a partition list prepared by the compiler and is used by the executor when it first opens the object, to support node and disk autonomy. If some of the nodes and volumes across which an object is partitioned are offline, the executor can attempt to open a partition on another node and volume given by an entry in the partition list. This default specifically controls the maximum number of volumes per node for which there will be an entry in the list. You can limit the number of partitions that the executor attempts to open per node by setting this default to a low value. Allowable values: 0 through 4294967295, SYSTEM. The default is 3.
GEN_MAX_NUM_PART_ NODE_ENTRIES	When a statically compiled statement references partitioned objects, this default is used to control the size of a partition list prepared by the compiler and used by the executor when it first opens the object, to support node and disk autonomy. If some of the nodes and volumes across which an object is partitioned are offline, the executor can attempt to open a partition on another node and volume given by an entry in the partition list. This default specifically controls the maximum number of nodes for which there will be one or more entry in the list. You can limit the number of nodes on which the executor attempts to find an available partition, by setting this default to a low value. Allowable values: 0 - 4294967295, SYSTEM. The default is 255.
PM_OFFLINE_ TRANSACTION_ GRANULARITY	Number of rows to be copied in an offline MODIFY transaction. This attribute enables partition operations, which can involve large amounts of data, to be done in many separate, smaller transactions. Allowable values: 50 to 4194303, inclusively. The default is 5000.
PM_ONLINE_ TRANSACTION_ GRANULARITY	Number of rows to be copied in an online MODIFY transaction. This attribute enables partition operations, which can involve large amounts of data, to be done in many separate, smaller transactions. Allowable values: 50 to 4194303, inclusively. The default is 400.

Attribute	Setting
POS_LOCATIONS	Physical locations of nonprimary partitions to be automatically created, specified as [*node*.]$*volume.* You can specify multiple locations separated by commas. If you enter a space (“ “), NonStop SQL/MX chooses the locations of the second through last partitions at random. NonStop SQL/MX does not place partitions on these types of disks: audit trail volumes, nonaudited disks, optical disks, phantom disks, or SMS virtual disks. See Creating Partitions Automatically on page 2-127. The default is no specification.
POS_NUM_OF_PARTNS	Number of partitions to be automatically created. If the value is greater than 1, NonStop SQL/MX creates that many partitions, including the primary partition. A value of 1 or 0 indicates that Partition Overlay Support (POS) is disabled. See Creating Partitions Automatically on page 2-127 for details. The default is 1.
POS_RAISE_ERROR	Determines whether an error should be raised when the POS feature cannot generate location names for the partitions to be created or if a warning should be raised indicating that the POS feature was not applied, and a simple table is created without partitions. When set to OFF (the default value,) a warning is displayed indicating that POS was not applied, and a simple table is created without partitions. When set to ON, an error is displayed indicating that location names could not be generated. The default is OFF.

Query Optimization and Performance

These attributes enable NonStop SQL/MX to optimize query execution:

Attribute	Setting
ALLOW_DP2_ROW_SAMPLING	Set to SYSTEM, ON, or OFF. When a SQL/MX query contains a SAMPLE clause, this attribute determines whether the sampling operation should be done by DP2 or by the SQL/MX Executor. When set to SYSTEM, the sampling is done by DP2 for sample percentages of up to 5%. When set to ON, the sampling is done by DP2 for sample percentages of up to 50%. When set to OFF, the sampling is done by the SQL/MX Executor and not by the DP2. For additional information about this setting, see the *SQL/MX Query Guide*. The default is SYSTEM.
ATTEMPT_ASYNCHRONOUS_ ACCESS	Set to ON or OFF. When set to ON, the optimizer generates plans that access multiple partitions asynchronously (that is, at the same time). With asynchronous access, the optimizer does not use ESPs to access the partitions in parallel. This setting also affects whether stream access to a table with partitions is allowed. See Stream Access Restrictions on page 2-348. For additional information about this setting, see the *SQL/MX Query Guide*. The default is ON.
ATTEMPT_ESP_PARALLELISM	Set to ON, OFF, or SYSTEM. If ON, the optimizer generates and considers plans that use ESP parallelism for all operators that can use ESP parallelism. If OFF, the optimizer never generates and considers plans that use ESP parallelism. If SYSTEM, the optimizer determines on an operator-by-operator basis when to generate and consider plans that use ESP parallelism. For additional information about this setting, see the *SQL/MX Query Guide*. The default is SYSTEM.

Attribute	Setting
CHECK_CONSTRAINT_PRUNING	Set to ON, OFF, RESET, or SYSTEM. The default is ON. If OFF, the constraint based query pruning optimization will not be tried on the subsequent queries. The CQD value RESET or SYSTEM sets the value of CQD back to the default value. The constraint based pruning uses the Constant Range Predicate Folding (CRPF) feature. The CRPF uses EncodedValue objects to store the actual values as `double` datatype with up to 15 digit of precision. For datatypes that have precision more than 15 digits (such as `largeint`), this conversion of actual value from more precision to `double`, causes comparisons to go wrong. Hence,constraint pruning is not applied when the value for comparison happens to have more than 15 digit of precision. For information on check constraint pruning feature, see the *SQL/MX Query Guide*.
CROSS_PRODUCT_CONTROL	Set to ON or OFF. ON reduces compile time by eliminating query plans that include unnecessary and expensive cross-products (joins without join predicates). For additional information about this setting, see the *SQL/MX Query Guide*. The default is ON.
DATA_FLOW_OPTIMIZATION	Set to ON or OFF. Reduces compile time by not considering some query plans that have relatively high data flow rates. The default is ON, resulting in improvement in compile time without impacting plan quality.
DDL_VIEW_SIMILARITY_CHECK	Specifies whether views by default are created with Similarity Check enabled or disabled. If specified, the SIMILARITY CHECK clauses in CREATE VIEW and ALTER VIEW statements take precedence over this CQD setting. The Similarity Check is not supported for Nested Views and views with VALUES clause. The CQD settings determine if an error is returned for these views. The following are the CQD settings and behavior: If set to ENABLE/SYSTEM, Similarity Check is enabled for supported view definitions, otherwise Similarity Check is disabled. If set to ON, Similarity Check is enabled for supported view definitions, otherwise the view is not created and an error is returned. DISABLE/OFF - Similarity Check is disabled.

Attribute	Setting
DP2_CACHE_4096_BLOCKS	Specifies the number of 4 KB blocks allocated for the disk cache. This value is used by the compiler to determine the cost of a table/index scan operator. You should set the value of this attribute to the average value of 4 KB block disk cache settings for all volumes in the system. The current value for the number of 4 KB blocks allocated to a disk cache can be determined by using SCF. See the *SCF Reference Manual for G-Series RVUs* for details on that product. Allowed values: 1 through 1,4294967295. The default is 1024.
FFDC_DIALOUTS_FOR_MXCMP	Set to ON or OFF. Controls whether FFDC dial-outs should occur when the compiler terminates abnormally or detects an internal error. The default is OFF, disallowing dial-outs.
GENERATE_EXPLAIN	Enables generation of EXPLAIN information at compile time. For MXCI, the default is automatically turned on by a CONTROL QUERY DEFAULT GENERATE_EXPLAIN 'ON' command issued by MXCI at startup time. For performance testing in MXCI, you might want to turn off GENERATE_EXPLAIN. You must explicitly turn off GENERATE_EXPLAIN if you do not want to include explain generation time while preparing statements from MXCI or while analyzing performance testing in MXCI. You must explicitly turn on GENERATE_EXPLAIN for NonStop MXCS and other embedded dynamic queries if you want to look at access plan or EXPLAIN information. The default setting is ON for embedded static queries and OFF for dynamic queries (from embedded programs or MXCS).
GEN_EIDR_BUFFER_SIZE	Combined with GEN_PA_BUFFER_SIZE, determines the buffer size for partition access operations. The two default settings must be equal. Each partition has one partition access operator and, by default, each partition access operator has 7 buffers associated with it. For OLTP applications, reducing buffer size to 4 KB can improve performance by reducing memory usage. For DSS applications, use the default. For additional information about this setting, see the *SQL/MX Query Guide*. The default buffer size is 31 KB.

Attribute	Setting
GEN_PA_BUFFER_SIZE	Combined with GEN_EIDR_BUFFER_SIZE, determines the buffer size for partition access operations. The two default settings must be equal. For OLTP applications, reducing buffer size to 4 KB can improve performance by reducing memory usage. For DSS applications, use the default. For additional information about this setting, see the *SQL/MX Query Guide*. The default buffer size is 31 KB.
INDEX_ELIMINATION_LEVEL	Set to MINIMUM, MEDIUM, or MAXIMUM to indicate the degree of heuristic elimination of indexes consideration by the optimizer. Elimination of less promising indexes results in improvement in compile time. MINIMUM value implies no elimination, and MAXIMUM implies maximum elimination. The default value is MAXIMUM.
JOIN_ORDER_BY_USER	Enables (ON) or disables (OFF) the join order specified you specify in the FROM clause of a query. When set to ON, the optimizer considers only execution plans that have the join order you specify. For additional information about this setting, see the *SQL/MX Query Guide*. The default is OFF.
MAX_ESPS_PER_CPU_PER_OP	Set to the maximum number of ESPs the optimizer considers starting for each CPU for a given operator. For additional information about this setting, see the *SQL/MX Query Guide*. The default is 1, which limits the optimizer to plans with only one ESP per CPU for a given operator. Allowable values: 1, >1.
MDAM_SCAN_METHOD	Enables (ON) or disables (OFF) the MultiDimensional Access Method (MDAM). In certain situations, the optimizer might choose MDAM inappropriately, causing poor performance. **SQL/MP Considerations:** SQL/MP users know this attribute as CONTROL TABLE MDAM ENABLE. The default is ON.

Attribute	Setting
MEMORY_USAGE_SAFETY_NET	Specifies the SQL/MX compiler memory threshold used for generating optimal query plan in megabytes (MB). For complex queries, when the optimizer memory reaches this threshold, plans are pruned to reduce the memory growth. Setting this attribute too low can result in sub-optimal plans. The actual memory available to a process on the NonStop operating system is limited to approximately 1.4 GB. Therefore, setting this value greater than 1.4 GB does not imply that the process will have memory larger than the system limit. Memory size is confined to the limits of the underlying operating system. When the optimizer reaches the set threshold, a warning 6020 is displayed and the query compiles successfully. The generated plan might not be optimal and execution might be slow. This attribute can have a value in the range 800 through 4096. The default value is OFF, which implies optimizer can use the maximum available memory.
MIN_MAX_OPTIMIZATION	Set to ON or OFF. This performance optimization enables the compiler to read only the result row or a select number of rows to answer minimum (MIN) or maximum (MAX) aggregate expressions. The compiler can perform this type of optimization only when the rows are naturally ordered on the MIN-MAX column. If OFF, this type of optimization is disabled. The default is ON.
MSCF_ET_REMOTE_MSG_ TRANSFER	The value of this default is used to factor in the cost of transferring messages to and from a remote node. It reflects the bandwidth of the physical communication link. You should set it to a value greater than the cost factor of transferring messages to or from a local node, which is 0.000046. Allowable values: 1.175494351e-38 through 3.402823466e+038. The default value is 0.00005.
MULTIUNION	Set to ON, OFF, RESET, or SYSTEM. The default is ON. When set to ON, NonStop SQL/MX generates a MultiUnion node. When set to OFF, NonStop SQL/MX does not generate a MultiUnion node. When set to RESET or SYSTEM, the default value is reset. For information on the MultiUnion operator, see the *SQL/MX Query Guide*.

Attribute	Setting
NUMBER_OF_USERS	Set to the number of users that can run concurrent queries that use large amounts of memory. For these queries, the optimizer uses this number to limit the amount of memory available for one user. The larger the number, the less memory available for operators (such as hash join) that use much memory. Allowed values: 1 through 1,4294967295. The default setting is 1, which means that all available memory can be assigned to one query.
OLT_QUERY_OPT	Set to ON or OFF. When set to ON, the NonStop SQL/MX enables a fast path evaluation method for certain simple SQL queries, such as a single table unique select. For additional information about this setting, see the *SQL/MX Query Guide*. The default is ON.
OPTIMIZATION_LEVEL	Set to 0, 2, 3, or 5 to indicate increasing effort in optimizing SQL queries. Values 1 and 4 are reserved for future use. For additional information about this setting, see the *SQL/MX Query Guide*. The default is 3.
OPTS_PUSH_DOWN_DAM	Set to ON (1) or OFF (0). When set to ON, the system considers pushing down a plan to DAM for compound statements or nested joins. When set to OFF, the system does not consider this option. When pushing a plan down to DAM is possible (the value is ON), NonStop SQL/MX might not select the push-down plan because of its cost. For additional information about this setting, see the *SQL/MX Query Guide*. The default is OFF.
PARALLEL_NUM_ESPS	Set to the keyword SYSTEM or to the maximum number of ESPs (an unsigned positive integer) that should be used for a particular operator. If set to SYSTEM, NonStop SQL/MX calculates the value. If set to a number, the value must be less than the number of CPUs in the cluster. For additional information about this setting, see the *SQL/MX Query Guide*. Allowable values: 1 through 2147483647. The default is SYSTEM (no maximum).

Attribute	Setting
PREFERRED_PROBING_ ORDER_FOR_NESTED_JOIN	Set to ON or OFF. If ON, the optimizer generates and considers plans where the rows of the inner table must be read in the key order of the access path. If OFF, the optimizer does not generate plans where the rows must be read in the key order of the access path. The default is OFF.
REMOTE_ESP_ALLOCATION	Set to ON, OFF, or SYSTEM. If ON, NonStop SQL/MX is forced to bring up ESPs on all the systems that are in the scope of the specific query and all target systems become active systems. If OFF, NonStop SQL/MX is forced to bring up all ESPs on the local system only. If SYSTEM, NonStop SQL/MX decides which target systems should be used for ESP placement. In this case, systems chosen as active are a subset of the target systems. The SYSTEM setting is the preferred setting for REMOTE_ESP_ALLOCATION. For additional information about this setting, see the *SQL/MX Query Guide*. The default is SYSTEM.
SORT_MAX_HEAP_SIZE_MB	The default value is used for allocating the heap memory size for operations involving the sort operator. The minimum and maximum values are 0 and 1024 respectively. The default value is 20.
UNION_TRANSITIVE_PREDICATES	Set to ON, OFF, RESET, or SYSTEM. When set to ON, NonStop SQL/MX generates transitive predicates for join on unions. When set to OFF, NonStop SQL/MX does not generate transitive predicates for join on unions. The default is ON.
UPD_ORDERED	Set to ON or OFF. If ON, the optimizer generates and considers plans where the rows must be inserted, updated, or deleted in clustering key order. If OFF, the optimizer does not generate plans where the rows must be inserted, updated, or deleted in clustering key order. The default is ON.
ZIG_ZAG_TREES	Set to ON or OFF. Enables (ON) or disables (OFF) the optimizer to consider zigzag trees in addition to linear trees. For additional information about this setting, see the *SQL/MX Query Guide*. The default is OFF.

Note. The CHECK_CONSTRAINT_PRUNING, MULTIUNION, and UNION_TRANSITIVE_PREDICATES CQDs are available only on systems running J06.08 and later J-series RVUs and H06.19 and later H-series RVUs.

For more information about query optimization, see the *SQL/MX Query Guide*.

Query Plan Caching

These attributes enable NonStop SQL/MX to cache query plans:

Attribute	Setting
QUERY_CACHE	Set to a value between 0 to 4194303. Indicates the size in kilobytes to which the cache is allowed to grow. The default setting is 1024, which activates a query cache that can grow to 1024 KB in the current session. To deactivate the query cache in the current session, set QUERY_CACHE to 0. If a query cache was allocated, this setting frees it. For additional information about this setting, see the *SQL/MX Query Guide*.
QUERY_CACHE_MAX_VICTIMS	Set to a value between 0 and 4194303. Indicates the maximum number of cache entries that can be displaced to accommodate a new entry and stay within the size limit of the cache. Setting this attribute to a very large value means that all the cache entries could be displaced to accommodate one very large query. Setting this attribute to 0 means that, when the cache becomes full, no cache entries (pinned or unpinned) can be displaced, and no new entries can be entered into the cache. For additional information about this setting, see the *SQL/MX Query Guide*. The default setting is 10 cache entries.

Attribute	Setting
QUERY_CACHE_REQUIRED_PREFIX_KEYS	Set to a value between 0 and 255. Specifies how many and which columns of a composite primary or partition key are required for an equality predicate to be considered cacheable. If the attribute is set to a value greater than the number of columns in a composite key, all columns of the key are required. The value 0 means that the presence of any one column of a composite primary or partition key in an equality key predicate is sufficient to make that predicate cacheable. A value *n* that is greater than 0 (zero) but less than the number of columns in the key indicates that the first *n* columns of the key must be present in a key predicate for that predicate to be considered cacheable. For additional information about this setting, see the *SQL/MX Query Guide*. The default setting is 255, which means that only complete primary or partition key equality predicates are cacheable. To avoid compromising query plan quality, You should keep the system-defined default setting of 255.
QUERY_CACHE_STATEMENT_PINNING	Set to ON, OFF, or CLEAR. Controls whether queries are entered into the cache as pinned or unpinned. You might have important, compile-time critical queries that you want to ensure are in the cache when needed. When a query is pinned in the cache, it usually cannot be displaced from the cache unless the cache becomes full of pinned queries. In this case, the least recently used pinned entries also become displaceable. The value CLEAR means that all subsequent query cache entries are unpinned, and all pinned entries in the cache are also unpinned. The value ON means that all subsequent query cache entries into the cache are pinned. The default setting, OFF, means that all subsequent query entries into the cache are unpinned.

For a full discussion of query plan caching, see the *SQL/MX Query Guide*.

Referential Action

This attribute determines how NonStop SQL/MX handles referential action in ALTER TABLE and CREATE TABLE statements:

Attribute	Setting
REF_CONSTRAINT_NO_ ACTION_LIKE_RESTRICT	Controls how NO ACTION *referential action* is treated. Set to OFF, SYSTEM, or ON. OFF means that SQL issues error 1301 SYSTEM means that SQL issues warning 1302 saying that it behaves like RESTRICT. ON means that NO ACTION behaves like RESTRICT, without warning or errors. SYSTEM is the default value.

Row Maintenance

These attributes determine how NonStop SQL/MX maintains rows in tables:

Attribute	Setting
INSERT_VSBB	Method of inserting rows into a table. Set to: OFF for simple inserts SYSTEM for DAM to determine the method USER to use VSBB LOADNODUP to insert with no check for duplicates The default is SYSTEM.
READONLY_CURSOR	Set to TRUE or FALSE. If set to TRUE, you must declare cursors with the FOR UPDATE clause for the named columns or all columns to be updatable. This setting improves cursor performance. If set to FALSE and the declarations omit FOR UPDATE or FOR READ ONLY, all columns are updatable. In SQL/MX, DELETE WHERE CURRENT OF does not work without the FOR UPDATE clause if READONLY_CURSOR is set to true. **SQL/MP Consideration**: DELETE WHERE CURRENT OF works without the FOR UPDATE clause. The default is TRUE.

Scratch Disk Management

These attributes determine how NonStop SQL/MX manages scratch disks for the sort operation:

Attribute	Setting
SCRATCH_DISKS	Set to a list of scratch disk volumes, where each item in the list has the form `[\node.]$volume`, and the items in the list are separated by a comma (`,`). Use this default to restrict scratch disks to the volumes specified. If none of the three scratch disk defaults are set, the system determines the scratch disk volumes to be used.
SCRATCH_DISKS_ EXCLUDED	Set to a list of scratch disk volumes, where each item in the list has the form `[\node.]$volume`, and the items in the list are separated by a comma (`,`). Use this default to exclude certain volumes from being used for scratch disks. If none of the three scratch disk defaults are set, the system determines the scratch disk volumes to be used.
SCRATCH_DISKS_ PREFERRED	Set to a list of scratch disk volumes, where each item in the list has the form `[\node.]$volume`, and the items in the list are separated by a comma (`,`). Use this default to indicate preference for volumes to be used for scratch disks. If none of the three scratch disk defaults are set, the system determines the scratch disk volumes to be used.
SCRATCH_FREESPACE_ THRESHOLD_PERCENT	Indicates how much free space, as a percentage, is left on a disk as a threshold. When that threshold is reached, hash or sort operations will use a different disk. If all disks reach their threshold, NonStop SQL/MX displays an error. The default value is 10. When disk usage reaches the point where only 10 percent of the space remains, hash or sort or operations stop using that disk.

About SQL/MX Scratch Disks

NonStop SQL/MX selects a disk to be used for a scratch file from the pool of available disks. The pool initially consists of the set of all suitable disks. Disks such as optical disks, phantom disks, and SMS virtual disks are not considered suitable. The disks specified by the SCRATCH_DISKS_EXCLUDED control are removed. If the SCRATCH_DISKS control is specified, the disks that are not specified in the SCRATCH_DISKS control are removed from the pool. From this disk pool, a disk is selected based on this criteria:

- The amount of used space on the disk. (rank * 30)
- The number of scratch files on the disk. (rank * 70)
- The number of fragments on the disk. (rank * 20)
- The biggest available fragment on the disk. (inverted rank * 80)
- Is the disk a preferred disk? (10000)
- Is the disk the primary disk of the CPU of this process? (100000)

The value in parentheses indicates the weighting of that criterion. The rank is the ordinal rank of that disk among all the disks in the pool based on the criterion. The inverted rank is the inverted ordinal rank. In the case of the biggest available fragment criterion, if the pool contains 20 disks, the disk with the biggest available fragment would have an inverted rank of 20. The weights are summed for all the disks in the pool, and the disk with the biggest weight is selected. As can be seen, the primary disk of the current CPU is given a large weight.

In NonStop SQL/MX, a scratch file can overflow to another disk. So, if a scratch file becomes full or if the disk becomes full, the operation does not necessarily fail. An additional scratch file on another disk is selected (using the criterion procedure). As a result, there is no 2 GB limit on scratch space.

In NonStop SQL/MX, the operations that can create scratch files are sort, hash join, and hash groupby. They all use the criterion procedure to determine which scratch disk to use.

NonStop SQL/MX does not manage swap file space directly. Instead, SQL/MX processes rely on the Kernel-Managed Swap Facility (KMSF), which is set up in the NonStop operating system with the NSKCOM tool. Each CPU has an associated swap file.

Sequence Functions

This attribute enables NonStop SQL/MX to optimize the execution of sequence functions:

Attribute	Setting
DEF_MAX_HISTORY_ ROWS	Number of rows the SEQUENCE BY operator keeps in its history buffer. This value affects sequence functions that examine a maximum number of rows and overrides any larger maximum specified as a sequence function argument. Allowed values: 1 through 2147483647. The default is 1024.

Statement Atomicity

These attributes affect NonStop SQL/MX's ability to undo the effects of an insert, update, or delete operation, when an error occurs during the operation, without having to abort the entire transaction:

Attribute	Setting
DOOM_USERTRANSACTION	Controls whether NonStop SQL/MX dooms a transaction when it encounters an unrecoverable error and the transaction cannot be rolled back to a savepoint. When a transaction is doomed by TMF, it is marked for abort and has to be aborted explicitly. A new transaction is started before the user can proceed. When set to ON and NonStop SQL/MX cannot roll back to a savepoint, it dooms the transaction. When set to OFF, NonStop SQL/MX dooms the transaction if it inherits the transaction from the user application or the JDBC or ODBC drivers. NonStop SQL/MX aborts the transaction if it started the transaction. The default is OFF.
UPD_ABORT_ON_ERROR	Controls whether an error that occurs during the performance of an insert, update, or delete causes an abort. ON means that NonStop SQL/MX will abort a user transaction after an error in an IUD statement. This behavior is similar to that of SQL/MX Release 1.8. OFF means that NonStop SQL/MX will not abort a user transaction after an error in an IUD statement. The default is OFF.
UPD_SAVEPOINT_ON_ERROR	Controls whether DP2 savepoints are to be used and whether the transaction is aborted in case of an error during an IUD statement. ON means that DP2 savepoints are used, if possible. OFF means that DP2 savepoints are not used and that the transaction will be aborted in case of an error. The default is ON.

Statement Recompilation

These attributes affect statement recompilation at execution time:

Attribute	Setting
AUTOMATIC_ RECOMPILATION	Set to ON or OFF. If set to ON, an SQL statement is automatically recompiled at run time, depending on the outcome of various factors. If OFF, NonStop SQL/MX does not recompile the statement and returns an error if various comparisons fail. The default is ON.

Attribute	Setting
INTERACTIVE_ACCESS	Set to ON or OFF. If set to ON, the compiler selects the most appropriate index-based access plan. If OFF, the compiler follows normal behavior and does not emphasize index-based access plans. The default is OFF.
RECOMPILE_ON_ PLANVERSION_ERROR	Set to ON or OFF. If set to ON, a SQL statement is automatically recompiled at run time in case of versioning errors. If OFF, NonStop SQL/MX does not recompile the statement and returns error 25302 or 25303. The default is ON.
RECOMPILATION_ WARNINGS	Set to ON or OFF. If set to ON, when a statement is automatically recompiled in an application (because of various factors), NonStop SQL/MX returns a warning message. A warning is also returned when a similarity check passes. Set this default to ON only to direct warning messages to an application when automatic recompilation take place or to notify it if a similarity check has passed. When automatic recompilation occurs, NonStop SQL/MX always logs an EMS event regardless of the setting of this CQD. The default is OFF.
SIMILARITY_CHECK	Set to ON or OFF. If set to ON, NonStop SQL/MX compares whether two tables used in an SQL statement (the previous compile-time table and the new run-time table) are sufficiently similar so that the previous access plan can be used for the new table. If OFF, NonStop SQL/MX automatically recompiles the statement, depending on the outcome of late name resolution, timestamp comparison, or table redefinition. The default is ON.

If you issue a CONTROL TABLE statement for the SIMILARITY_CHECK option, the specified control table value overrides the system-defined default setting. See CONTROL TABLE Statement on page 2-74.

For more information about late name resolution, similarity checks, and automatic recompilation, see the *SQL/MX Programming Manual for C and COBOL*.

Stored Procedures in Java

This attribute specifies the Java Virtual Machine (JVM) startup options for the Java environment of a stored procedure in Java (SPJ):

Attribute	Setting
UDR_JAVA_OPTIONS	Set the attribute value to one or more Java options within single quotes (for example, '*java-option1 java-option2*'). Each Java option can be any Java option supported by the HP NonStop Server for Java. The Java options must conform to Java syntax, contain no embedded white space, and have a single space separating each option. If the same option is specified more than once, the JVM allows the last occurrence in the string to take precedence. CALL statements compiled with this setting are serviced in an SPJ environment that uses the specified JVM startup options. Set to OFF to specify no application-specific JVM startup options in the SPJ environment. CALL statements compiled with this setting are serviced in an SPJ environment that does not use application-specific JVM startup options. Set to ANYTHING to enable NonStop SQL/MX to chose the JVM startup options for an SPJ environment. NonStop SQL/MX does not guarantee the types of JVM startup options that are used in a particular SPJ environment. CALL statements compiled with this setting are serviced in an SPJ environment chosen by NonStop SQL/MX. The default is OFF.

For more information on the supported Java options, see the *NonStop Server for Java Tools Reference Pages*. For more information on how to use the UDR_JAVA_OPTIONS default attribute, see the *SQL/MX Guide to Stored Procedures in Java*.

Stream Access

These attributes enable NonStop SQL/MX to implement the queuing and publish/subscribe services:

Attribute	Setting
MATERIALIZE	Controls whether inner tables (the non-streamed tables) of join operations between streams and base tables are materialized. Set to ON, OFF, or SYSTEM to direct the SQL compiler to materialize inner tables for join operations (ON), not materialize inner tables (OFF), or allow the system to determine whether to materialize inner tables (SYSTEM). If you want changes to the inner table that are made while the stream is active to be visible in the join, set this value to OFF. The default is SYSTEM.

Attribute	Setting
STREAM_TIMEOUT	The time in hundredths of seconds for a cursor fetch operation using stream access to wait for more rows before timing out. Setting this default directs NonStop SQL/MX to not wait for more rows beyond the specified time but to return with error code 8006. This default is valid for compile-time stream timeout. For run-time stream timeout, see SET TABLE TIMEOUT Statement on page 2-372. Using a low value can result in a timeout before all rows are returned (due to delays between processes). If you use a low value (for example, 300), the application unblocks and either closes the cursor to not wait any longer or retrieves the fetch. Any value < 0 directs NonStop SQL/MX to wait indefinitely until there are no more rows to return. Setting it to RESET changes it back to the value in effect at the start of the session. Allowable values: -2147483648 through 2147483647. The default is -1.

For more information about these defaults, see *SQL/MX Queuing and Publish/Subscribe Services*.

Table Management

These attributes enable NonStop SQL/MX to manage tables:

Attribute	Setting
DEFAULT_BLOCKSIZE	Enables you to specify the default behavior when database objects are created that do not specify a BLOCKSIZE. Valid values are: • 4096— for database objects created without explicitly stating the BLOCKSIZE attribute, the block size will be 4 KB. This is the system default value. • 32768— for database objects created without explicitly stating the BLOCKSIZE attribute, the block size will be 32 KB.

Attribute	Setting

SAVE_DROPPED_TABLE _DDL	Controls whether definitions of dropped tables are saved to enable them to be recovered.
	If set to ON, DDL information for a dropped table is saved to a file called *`catalog.schema.tablename`-yyyymmdd-`timestamp`*.ddl in the OSS directory /usr/tandem/sqlmx/ddl.
	For example, if a table called CAT.SCH.T123 is dropped at 12:53:31 PM on July 29, 2003, the full OSS path name of the saved DDL file would be:
	
	`/usr/tandem/sqlmx/ddl/CAT.SCH.T123-20030729-125331.ddl`
	
	To drop the table you must have write access to this directory or you receive error 1232:
	*** ERROR[1232] A file error occurred when saving dropped table DDL for table *`table`* to /usr/tandem/sqlmx/ddl.
	If the table name contains a delimited identifier, characters that are not permitted in OSS file names are replaced by underscores. Quotes delimiting the identifier are removed.
	For example, if the table CAT."S&C%H"."T*A*B?01" is dropped at 12:57:15 am on April 24, 2003, the saved DDL file would be:
	
	`/usr/tandem/sqlmx/ddl/CAT.S_C_H.T_A_B_01-20030424-215715.ddl`
	
	If the 3-part ANSI name exceeds the maximum OSS file name length of 248, it is truncated to 248 characters.
	Despite similarities in the resulting file names because of character replacement or file name truncation, the files are always distinguishable by the trailing timestamp portion of the name and the contents of the file, which always indicates the full ANSI name of the table.
	
	NonStop SQL/MX does not remove saved DDL files. You must remove unwanted files from this location. If you do not periodically remove these files, the OSS directory will become full and DROP TABLE will no longer succeed. Database administrators should monitor the saved DDL location /usr/tandem/sqlmx/ddl for the accumulation of unneeded files.
	Here is an example of a script that will delete DDL files every seven days:
	
	`find  /usr/tandem/sqlmx/ddl  -mtime +7`
	`-print  |  grep "/ddl/" | sed "s/./rm &/"`
	`| sh`
	
	In development and testing environments where tables are frequently created and dropped it is recommended that this value be set to OFF.
	The default is ON.

Attribute	Setting
VARCHAR_PARAM_ DEFAULT_SIZE	Depending on context, untyped parameters might be converted to the VARCHAR type during compilation of a query. The default length for this type is 255 characters. This CQD allows you to change the default length. Allowable values: 1 to 32768. The default is 255 characters.

Trigger Management

This attribute enables NonStop SQL/MX to manage trigger temporary tables:

Attribute	Setting

TEMPORARY_TABLE_ HASH_PARTITIONS	Describes the partitioning scheme for trigger temporary tables by listing volumes across which the temporary tables can be hash partitioned, specified as *[\node.]$volume*, enclosed in single quotes. You can specify multiple locations separated by commas or colons. These examples are all valid:
	`Control query default TEMPORARY_TABLE_HASH_PARTITIONS '$data01,$data02,$data03';`
	`Control query default TEMPORARY_TABLE_HASH_PARTITIONS '$data01:$data02:$data03';`
	
	`Control query default TEMPORARY_TABLE_HASH_PARTITIONS '$data01,$data02:$data03';`
	If you specify more than one volume, the temporary table is hash partitioned over all those partitions. Range partitioning is not supported.
	If no system default is specified, NonStop SQL/MX uses the default location of the creator of the first trigger. If the default is changed, the change affects temporary tables created after the change. Previously created temporary tables will retain the previous setting. The partitioning scheme of the trigger subject table is unrelated to the temporary table.

Examples of SYSTEM_DEFAULTS Table

- Insert a row into the SYSTEM_DEFAULTS table to set the current default setting for the transaction isolation level:

```
INSERT INTO SYSTEM_DEFAULTS
  (ATTRIBUTE, ATTR_VALUE)
  VALUES ('ISOLATION_LEVEL', 'SERIALIZABLE');
```

- Query the SYSTEM_DEFAULTS table to obtain the current default setting for the transaction isolation level:

```
SELECT ATTRIBUTE, ATTR_VALUE FROM SYSTEM_DEFAULTS
  WHERE ATTRIBUTE = 'ISOLATION_LEVEL';
```

- Set a new value for the transaction isolation level:

```
UPDATE SYSTEM_DEFAULTS
  SET ATTR_VALUE = 'READ COMMITTED'
  WHERE ATTRIBUTE = 'ISOLATION_LEVEL';
```

- Set the level of optimization for the next query to be executed. The CONTROL QUERY DEFAULT statement does not change the settings of the SYSTEM_DEFAULTS table.

```
CONTROL QUERY DEFAULT OPTIMIZATION_LEVEL '0';
```

User Metadata Tables (UMD): Histogram Tables

A histogram is a representation of a relationship in which each value of some dependent variable corresponds to a range of values of the associated independent variable or variables. For example, a histogram might be a chart showing the number of people in New York in various age ranges.

NonStop SQL/MX provides a method for generating histograms that show how data is distributed with respect to a column or a group of columns within a table. The purpose of generating these statistics is to enable the optimizer to create efficient access plans.

When generating a histogram for a table, NonStop SQL/MX divides the range of specified column values of the table into some number of intervals, distributing the rows evenly within these intervals. It computes statistics associated with each interval and then uses the statistics to devise optimized plans.

You can use the statistics in the histogram tables as a basis for partitioning large tables. For example, if the employee number (the EMPNUM column, which is the primary key) in the EMPLOYEE table has a nonuniform distribution, use the histogram statistics to divide the range of employee numbers into partitions that distribute rows evenly. See Examples of Histogram Tables on page 10-92.

Creating Histogram Tables

NonStop SQL/MX automatically creates histogram tables when a schema is created: HISTOGRAMS and HISTOGRAM_INTERVALS for SQL/MX tables. HISTINTS and HISTOGRM SQL/MP tables are automatically created and registered in the catalog of the primary partition of the user table you specify in the UPDATE STATISTICS statement, if they do not already exist for that table.

Before you drop the SQL/MP catalog that contains the histogram tables, you must explicitly drop both of the tables.

You cannot update statistics on system metadata tables, including tables residing in the DEFINITION_SCHEMA, MXCS_SCHEMA, SYSTEM_DEFAULTS_SCHEMA, and SYSTEM_SCHEMA.

Generating Histogram Statistics

The UPDATE STATISTICS statement generates logical (table and column level) statistics that are stored in histogram user tables. To examine the current statistics, use the SELECT statement. The histograms for user tables registered in the same catalog reside in the same HISTOGRAMS and HISTOGRAM_INTERVAL or HISTINTS and HISTOGRM tables.

You can specify the number of intervals for the table statistics in the UPDATE STATISTICS statement. If you do not specify the number of intervals, NonStop SQL/MX provides a default number based on the table size and other factors.

The histogram tables are not automatically updated when you alter a table for which statistics are stored. Therefore, after you alter a table, you should execute UPDATE STATISTICS again for the table to keep its histogram statistics current.

If you drop an SQL/MP user table with DROP TABLE, the obsolete histograms for that table are not immediately deleted in the histogram tables. You can use the CLEAR option in UPDATE STATISTICS to delete all histograms for that table before you drop the table. See UPDATE STATISTICS Statement on page 2-402. Obsolete rows in SQL/MX histogram tables are automatically deleted.

Before you update statistics you can estimate the size of SQL/MX and SQL/MP temporary tables, in rows, based on the base tables size in rows. For example, if the base table has 12 million rows and you request a 2% sample, the temporary table will have approximately 240,000 rows.

Histogram Table Properties

	SQL/MX Objects	**SQL/MP Objects**
Histogram tables	Registered in the same catalog.schema as the table.	Registered in the catalog of the primary partition of the table.
	Located in the same catalog.schema as the table.	Located in the same \node.$vol.subvol as the catalog.
	File names: *catalog.schema*.HISTOGRAMS *catalog.schema*.HISTOGRAM_INTERVALS	File names: *\node.$vol.subvol*.HISTOGRM *\node.$vol.subvol*.HISTINTS
Temporary tables	Registered in the same catalog.schema as the table.	Registered in the catalog of the primary partition of the table.
	Located in a volume randomly chosen by NonStop SQL/MX from the list specified by HIST_SCRATCH_VOL, or in the volume specified in the _DEFAULTS define. If multiple volumes are specified with HIST_SCRATCH_VOL and the table does not contain a system generated key (SYSKEY), hash partitioning is performed over all specified volumes. May be audited or non-audited.	Located in a volume randomly chosen by NonStop SQL/MP from the list specified by HIST_SCRATCH_VOL, or in the same \node.$vol as the primary partition, in the ZZMXTEMP subvolume. Single partition. May be audited or non-audited.
	File name: *catalog.schema*.SQLMX_*tablename*	File name: *\node.$vol*.ZZMXTEMP.*tablename*

SQL/MX Objects	SQL/MP Objects
Size limits: Because files are always format 2, the temporary table is limited to 1 TB or the amount of available space on each volume.	Size limits: File format is determined by format of the base table's primary partition. If it is format 1, the temporary table is limited to 2 GB. If it is format 2, the temporary table is limited to 1 TB or the amount of available space on the disk volume.

HISTOGRAMS Table

The SQL/MX HISTOGRAMS table describes columns, interval count, total number of rows and number of unique rows, and the low and high values of column distribution for a table:

Column Name	Data Type	Description
*1 TABLE_UID	LARGEINT	The UID of the table for which this histogram is generated.
*2 HISTOGRAM_ID	INT UNSIGNED	System-generated ID for the histogram. Each HISTOGRAM_ID has a corresponding ID in the HISTOGRAM_INTERVALS table.
*3 COL_POSITION	INT	Column position in a column group for which the histogram is generated.
4 COLUMN_NUMBER	INT	Table column number for which this histogram is generated.
5 COLCOUNT	INT	Number of columns in the column group.
6 INTERVAL_COUNT	SMALLINT	Number of intervals in the histogram. If the value is n, there are n+1 corresponding rows in the HISTOGRAM_INTERVALS table with the same HISTOGRAM_ID.
7 ROWCOUNT	LARGEINT	Total number of rows in the table.
8 TOTAL_UEC	LARGEINT	Total number of unique entries in the table.
9 STATS_TIME	TIMESTAMP(0)	Start time of statistics generation, expressed as Greenwich mean time.
10 LOW_VALUE	VARCHAR(250) CHARACTER SET UCS2	Low value of column distribution (for the entire table).
11 HIGH_VALUE	VARCHAR(250) CHARACTER SET UCS2	High value of column distribution (for the entire table).
12 READ_TIME	TIMESTAMP(0)	A recent time for which this histogram is read.
13 READ_COUNT	SMALLINT	The number of times READ_TIME is updated since this histogram was last generated.

Column Name	Data Type	Description
14 SAMPLE_SECS	LARGEINT	Number of seconds required to create and populate sample table in seconds with the minimum value being 1 sec. If sampling is not used, this column is set to 0.
15 COL_SECS	LARGEINT	Number of seconds required to create statistics from column data for histogram. This column does not include sample time.
16 SAMPLE_PERCENT	SMALLINT	The SAMPLE_PERCENT is calculated using the following formula: `value_given_by_user` X 100 The value can range from 0-10000.
17 CV	FLOAT	The coefficient of variation. This is a value >=0 that represents the distribution of three occurrences of each distinct value.
18 REASON	CHAR(1)	Indicates why this histogram was last created. M created via manual run of update statistics I automatic initial creation based on request by optimizer N automatic regeneration, recently required by optimizer ' ' histogram is not generated
19 V1	LARGEINT	Reserved for future use
20 V2	LARGEINT	Reserved for future use
21 V3	LARGEINT	Reserved for future use
22 V4	LARGEINT	Reserved for future use
23 V5	VARCHAR(250) CHARACTER SET UCS2	Reserved for future use
24 V6	VARCHAR(250) CHARACTER SET UCS2	Reserved for future use

* Indicates primary key

HISTOGRAM_INTERVALS Table

The SQL/MX HISTOGRAM_INTERVALS table describes for each interval, the number of rows and number of unique rows in the interval and the value of the upper boundary for the interval:

Column Name	Data Type	Description
*1 TABLE_UID	LARGEINT	The UID of the table for which this histogram is generated.
*2 HISTOGRAM_ID	INT UNSIGNED	System-generated ID for the histogram. Each HISTOGRAM_ID has a corresponding ID in the HISTOGRAMS table.
*3 INTERVAL_NUMBER	SMALLINT	Sequence number for this interval.
4 INTERVAL_ROWCOUNT	LARGEINT	Number of rows in this interval.
5 INTERVAL_UEC	LARGEINT	Number of unique entries in this interval.
6 INTERVAL_BOUNDARY	VARCHAR(250) CHARACTER SET UCS2\	The value of the upper boundary for this interval.
7 STD_DEV_OF_FREQ	NUMERIC(12,3)	The standard deviation of F, where F is the set of {f1, … fn}, fi is the # of occurrences of value i in the interval, and n is the UEC of the interval
8 V1	LARGEINT	Reserved for future use
9 V2	LARGEINT	Reserved for future use
10 V3	LARGEINT	Reserved for future use
11 V4	LARGEINT	Reserved for future use
12 V5	VARCHAR(250) UCS2 COLLATE	Reserved for future use
13 V6	VARCHAR(250) UCS2 COLLATE	Reserved for future use

* Indicates primary key

HISTOGRM Table

The SQL/MP HISTOGRM table is a user table registered in the catalog of the primary partition of the table specified in the UPDATE STATISTICS statement that created the histogram tables. It describes columns, interval count, total number of rows and unique rows, and the low and high values of column distribution for the table for which the histogram is created.

△ **Caution.** HISTOGRM is an SQL/MP user table with the security of the user who runs the UPDATE STATISTICS command on the user tables of a particular SQL/MP catalog. It does not have the same protection as the SQL/MP catalog tables, which can be modified by licensed processes only. As such, system users who have write access to the table could enter invalid data, which could affect the performance or operation of NonStop SQL/MX. Therefore, you should secure access to the HISTOGRM table to a restricted group of users.

Column Name	Data Type	Description
*1 TABLE_UID	LARGEINT	The UID of the table for which this histogram is generated.
*2 HISTOGRAM_ID	INT UNSIGNED	System-generated ID for the histogram. Each HISTOGRAM_ID has a corresponding ID in the HISTINTS table.
*3 COL_POSITION	INT	Column position in a column group for which the histogram is generated. For example, columns in the group (a, b, c) have the corresponding positions of 0, 1, and 2.
4 COLUMN_NUMBER	INT	Table column number for which this histogram is generated.
5 COLCOUNT	INT	Number of columns in the column group.
6 INTERVAL_COUNT	SMALLINT	Number of intervals in the histogram. If the value is *n*, there are *n* +1 corresponding rows in the HISTINTS table with the same HISTOGRAM_ID.
7 ROWCOUNT	LARGEINT	Total number of rows in the table.
8 TOTAL_UEC	LARGEINT	Total number of unique entries in the table.
9 STATS_TIME	TIMESTAMP(0)	Start time of statistics generation (expressed as Greenwich mean time).
10 LOW_VALUE	VARCHAR(500)	Low value of column distribution (for the entire table).
11 HIGH_VALUE	VARCHAR(500)	High value of column distribution (for the entire table).

* Indicates primary key

HISTINTS Table

The SQL/MP table HISTINTS table is a user table registered in the catalog of the primary partition of the table specified in the UPDATE STATISTICS statement that created the histogram tables. It describes, for each interval of the table for which the histogram is created, the number of rows and unique rows in the interval and the value of the interval upper boundary.

△ **Caution.** HISTINTS is an SQL/MP user table with the security of the user who runs the UPDATE STATISTICS command on the user tables of a particular SQL/MP catalog. It does not have the same protection as the SQL/MP catalog tables, which can be modified by licensed processes only. As such, system users who have write access to the table could enter invalid data, which could affect the performance or operation of NonStop SQL/MX. Therefore, you should secure access to the HISTINTS table to a restricted group of users.

Column Name	Data Type	Description
*1 TABLE_UID	LARGEINT	The UID of the table for which this histogram is generated.
*2 HISTOGRAM_ID	INT UNSIGNED	System-generated ID for the histogram. Each HISTOGRAM_ID has a corresponding ID in the HISTOGRM table.
*3 INTERVAL_NUMBER	SMALLINT	Sequence number for this interval.
4 INTERVAL_ROWCOUNT	LARGEINT	Number of rows in this interval.
5 INTERVAL_UEC	LARGEINT	Number of unique entries in this interval.
6 INTERVAL_BOUNDARY	VARCHAR(500)	The value of the upper boundary for this interval.

* Indicates primary key

Examples of Histogram Tables

- Update the histogram statistics on the EMPNUM column for the EMPLOYEE table:

```
UPDATE STATISTICS FOR TABLE persnl.employee ON (empnum);

--- SQL operation complete.
```

- Use the SELECT statement to retrieve the statistics in the SQL/MX HISTOGRAMS table generated by the UPDATE STATISTICS statement, based on table and column names:

```
SELECT O.OBJECT_NAME
       TABLE_UID,
       C.COLUMN_NAME,
       H.HISTOGRAM_ID,
       H.INTERVAL_COUNT,
       H.ROWCOUNT,
       H.TOTAL_UEC,
       LOW_VALUE,
       HIGH_VALUE
  FROM cat.DEFINITION_SCHEMA_VERSION_1200.OBJECTS O,
       cat.DEFINITION_SCHEMA_VERSION_1200.COLS C,
       cat.sch.HISTOGRAMS H
  WHERE O.OBJECT_UID = H.TABLE_UID
    AND O.OBJECT_UID = C.OBJECT_UID
    AND C.COLUMN_NUMBER = H.COLUMN_NUMBER;
```

- Use the SELECT statement to retrieve the statistics in the SQL/MP HISTOGRM table generated by the preceding UPDATE STATISTICS statement, based on table and column names:

```
SELECT T.TABLENAME,
       H.TABLE_UID,
       C.COLNAME,
       H.HISTOGRAM_ID,
       H.INTERVAL_COUNT,
       H.ROWCOUNT,
       H.TOTAL_UEC,
       H.LOW_VALUE,
       H.HIGH_VALUE
  FROM $data06.mycat.TABLES T,
       $data06.mycat.COLUMNS C,
       $data06.mycat.HISTOGRM H
  WHERE T.CREATETIME = H.TABLE_UID
    AND T.TABLENAME = C.TABLENAME
    AND H.COLUMN_NUMBER = C.COLNUMBER;
```

- Use the SELECT statement to retrieve the statistics in the HISTINTS table generated by the preceding UPDATE STATISTICS statement (the columns are arranged horizontally):

```
SELECT * FROM HISTINTS;

TABLE_UID              HISTOGRAM_ID  INTERVAL_NUMBER
INTERVAL_ROWCOUNT      INTERVAL_UEC          INTERVAL_BOUNDARY
--------------------   ------------  ------------
--------------------   --------------------  ---------------
  211813264440965573      10036622             0
                   0                     0  (1)
  211813264440965573      10036622             1
                   4                     4  (32)
  211813264440965573      10036622             2
                   4                     4  (72)
  211813264440965573      10036622             3
                   3                     3  (89)
  211813264440965573      10036622             4
                   3                     3  (109)
  211813264440965573      10036622             5
                   3                     3  (180)
  211813264440965573      10036622             6
                   3                     3  (203)
  211813264440965573      10036622             7
                   3                     3  (207)
  211813264440965573      10036622             8
                   3                     3  (210)
  211813264440965573      10036622             9
                   3                     3  (213)
  211813264440965573      10036622            10
                   3                     3  (216)
  211813264440965573      10036622            11
                   3                     3  (219)
  211813264440965573      10036622            12
                   3                     3  (222)
  211813264440965573      10036622            13
                   3                     3  (225)
  211813264440965573      10036622            14
                   3                     3  (228)
  211813264440965573      10036622            15
                   3                     3  (231)
  211813264440965573      10036622            16
                   3                     3  (234)
  211813264440965573      10036622            17
                   3                     3  (337)
  211813264440965573      10036622            18
                   3                     3  (568)
  211813264440965573      10036622            19
                   3                     3  (992)
  211813264440965573      10036622            20
                   3                     3  (995)

--- 21 row(s) selected.
```

The data in the EMPLOYEE table is not distributed evenly with respect to the primary key because most of the rows in this table are in the EMPNUM equal to 200 range. The SQL/MX optimizer uses the histogram statistics, which divide the rows into intervals with approximately the same number of rows, to devise plans for efficient data access. You can use the boundary and row count information in the histogram tables to specify boundaries for physical partitions of large tables.

MXCS Metadata Tables

ASSOC2DS Table

ASSOC2DS is a metadata table in NONSTOP_SQL_*nodename*.MXCS_SCHEMA that associates the MXCS service to a data source:

Column Name	Data Type	Description
*1 ASSOC_ID	INT	Unique identifier for association service description.
*2 DS_ID	INT	Unique identifier for datasource description.
3 AUTOMATION	SMALLINT	0 data source is started manually. 1 data source is started automatically.
4 CURRENT_STATUS	SMALLINT	Current state of a data source defined for this association services: 0 STOPPED 1 STOPPING 2 STARTING 3 STARTED
5 DEFAULT_TYPE	SMALLINT	System default MXCS service.
6 LAST_STATUS_CHANGE	TIMESTAMP(6)	Julian timestamp of last time change to the data source status.
7 LAST_UPDATED	TIMESTAMP(6)	Julian timestamp of the last update to this description.

* Indicates primary key

DATASOURCES Table

DATASOURCES is a metadata table in NONSTOP_SQL_*nodename*.MXCS_SCHEMA that contains data source information:

Column Name	Data Type	Description
*1 DS_ID	INT	Unique identifier for this datasource description.
2 MAX_SRVR_CNT	INT	Maximum number of server instances that must be kept available for association requests.
3 AVAIL_SRVR_CNT	INT	Minimum number of server instances that must be kept available for association requests.
4 INIT_SRVR_CNT	INT	Minimum number of server instances that must be kept available for association requests.
5 START_AHEAD	INT	Number of server instances that are started ahead each time the number of available servers is less than the AVAIL_SRVR_CNT.
6 SRVR_IDLE_TIMEOUT	LARGEINT	Count in seconds indicating the length of time a server is allowed to sit idle (no client activity) before the server initiates disconnect.
7 CONN_IDLE_TIMEOUT	LARGEINT	Length of time in seconds that a connection request is allowed to wait before completing the connection: -1 wait forever. 0 do not wait. If a server instance is not immediately available, refuse connection requests.
8 REFRESH_RATE	LARGEINT	Frequency in seconds at which accumulated statistics are written to the permanent SQL statistics table(s).
9 LAST_UPDATED	TIMESTAMP(6)	Julian timestamp of last update to this description.

* Indicates primary key

ENVIRONMENTVALUES Table

ENVIRONMENTVALUES is a metadata table in NONSTOP_SQL_*nodename*.MXCS_SCHEMA that sets, controls, and defines environment data:

Column Name	Data Type	Description
*1 ENV_ID	INT	Identifier providing link to a parent object (data source, user or profile)
*2 VARIABLE_SEQUENCE	SMALLINT	Sort of environment variable based on type
*3 VARIABLE_TYPE	SMALLINT	Type of the environment variable: 0 SET 1 CONTROL 2 DEFINE
4 VARIABLE_VALUE	VARCHAR(3900)	Value of environment variable
5 LAST_UPDATED	TIMESTAMP(6)	Julian timestamp of last update to this description

* Indicates primary key

NAME2ID Table

NAME2ID is a metadata table in NONSTOP_SQL_*nodename*.MXCS_SCHEMA that associates service or data source names to an ID:

Column Name	Data Type	Description
*1 OBJ_ID	INT	Unique identifier
2 OBJ_TYPE	SMALLINT	Unique identifier
3 OBJ_NAME	VARCHAR(128)	SQLI_identifier, logical reference to object
4 VARIABLE_VALUE	VARCHAR(3900)	Reserved for future use
5 LAST_UPDATED	TIMESTAMP(6)	Julian timestamp of last update to this description

* Indicates primary key

RESOURCEPOLICIES Table

RESOURCEPOLICIES is a metadata table in NONSTOP_SQL_*nodename*.MXCS_SCHEMA that contains governing information.

Column Name	Data Type	Description
*1 RES_ID	INT	Identifier providing link to a parent object (datasource, user or profile)
2 ATTRIBUTE_NAME	VARCHAR(128)	Governed resource
3 LIMIT_VALUE	LARGEINT	Resource attribute test limit value
4 ACTION_ID	LARGEINT	Action to be taken when limit is reached
5 SETTABLE	SMALLINT	0 User’s preference values cannot override configured values 1 User's preference values can override configured values
6 LAST_UPDATED	TIMESTAMP(6)	Julian timestamp of last update to this status record

* Indicates primary key

A Quick Reference

This appendix provides a quick, alphabetic reference to commands, statements, and utilities. For other topics, see the Index.

A

B

C

D

E

H

I

L

M

O

P

B Reserved Words

The words listed in this appendix are reserved for use by NonStop SQL/MX. To prevent syntax errors, avoid using these words as identifiers in NonStop SQL/MP and in NonStop SQL/MX. In NonStop SQL/MX, if a Guardian name contains a reserved word, you must enclose the reserved word in double quotes (") to access that column or object. See Using SQL/MX Reserved Words in SQL/MP Names on page 6-57.

In these lists, an asterisk (*) indicates reserved words that are SQL/MX extensions. Words reserved by the ANSI standard are not marked.

Note. In SQL/MX Release 2.x, ABSOLUTE, DATA, EVERY, INITIALIZE, OPERATION, PATH, STATE, STATEMENT, STATIC, and START are not reserved words.

Reserved SQL/MX and SQL/MP Identifiers

SQL/MX treats these words as reserved when they are part of SQL/MX text or SQL/MP stored text. They cannot be used as identifiers unless you enclose them in double quotes. SQL/MP stored text is SQL text that NonStop SQL/MX retrieves from the SQL/MP catalog while processing SQL/MX text. SQL/MP stored text includes views, constraints, column defaults, first keys, clustering keys, and partitioning keys.

Table B-1. Reserved SQL/MX and SQL/MP Identifiers (page 1 of 4)

ACTION	FOR	PROTOTYPE*
ADD	FOREIGN	PUBLIC
ADMIN	FOUND	READ
AFTER	FRACTION*	READS
AGGREGATE	FREE	REAL
ALIAS	FROM	RECURSIVE
ALL	FULL	REF
ALLOCATE	FUNCTION	REFERENCES
ALTER	GENERAL	REFERENCING
AND	GET	RELATIVE
ANY	GLOBAL	REPLACE*
ARE	GO	RESIGNAL*
ARRAY	GOTO	RESTRICT
AS	GRANT	RESULT
ASC	GROUP	RETURN
ASSERTION	GROUPING	RETURNS
ASYNC*	HAVING	REVOKE
AT	HOST*	RIGHT

Table B-1. Reserved SQL/MX and SQL/MP Identifiers (page 2 of 4)

AUTHORIZATION	HOUR	ROLE
AVG	IDENTITY	ROLLBACK
BEFORE	IF*	ROLLUP
BEGIN	IGNORE	ROUTINE
BETWEEN	IMMEDIATE	ROW
BINARY	IN	ROWS
BIT	INDICATOR	SAVEPOINT
BIT_LENGTH	INITIALLY	SCHEMA
BLOB*	INNER	SCOPE
BOOLEAN	INOUT	SCROLL
BOTH	INPUT	SEARCH
BREADTH	INSENSITIVE	SECOND
BY	INSERT	SECTION
CALL*	INT	SELECT
CASE	INTEGER	SENSITIVE*
CASCADE	INTERSECT	SESSION
		SEQUENCE
CASCADED	INTERVAL	SESSION_USER
CAST	INTO	SET
CATALOG	IS	SETS
CHAR	ISOLATION	SIGNAL*
CHAR_LENGTH	ITERATE	SIMILAR*
CHARACTER	JOIN	SIZE
CHARACTER_LENGTH	KEY	SMALLINT
CHECK	LANGUAGE	SOME
CLASS	LARGE	SPACE
CLOB	LAST	SPECIFIC
CLOSE	LATERAL	SPECIFICTYPE
COALESCE	LEADING	SQL
COLLATE	LEAVE*	SQL_CHAR*
COLLATION	LEFT	SQL_DATE*
COLUMN	LESS	SQL_DECIMAL*
COMMIT	LEVEL	SQL_DOUBLE*
COMPLETION	LIKE	SQL_FLOAT*
CONNECT	LIMIT	SQL_INT*

Table B-1. Reserved SQL/MX and SQL/MP Identifiers (page 3 of 4)

CONNECTION	LOCAL	SQL_INTEGER*
CONSTRAINT	LOCALTIME	SQL_REAL*
CONSTRAINTS	LOCALTIMESTAMP	SQL_SMALLINT*
CONSTRUCTOR	LOCATOR	SQL_TIME*
CONTINUE	LOOP*	SQL_TIMESTAMP*
CONVERT	LOWER	SQL_VARCHAR*
CORRESPONDING	MAP	SQLCODE
COUNT	MATCH	SQLERROR
CREATE	MAX	SQLEXCEPTION
CROSS	MIN	SQLSTATE
CUBE	MINUTE	SQLWARNING
CURRENT	MODIFIES	STRUCTURE
CURRENT_DATE	MODIFY	SUBSTRING
CURRENT_PATH	MODULE	SUM
CURRENT_ROLE	MONTH	SYSTEM_USER
CURRENT_TIME	NAMES	TABLE
CURRENT_TIMESTAMP	NATIONAL	TEMPORARY
CURRENT_USER	NATURAL	TERMINATE
CURSOR	NCHAR	TEST*
CYCLE	NCLOB	THAN*
DATE	NEW	THEN
DATETIME*	NEXT	THERE*
DAY	NO	TIME
DEALLOCATE	NONE	TIMESTAMP
DEC	NOT	TIMEZONE_HOUR
DECIMAL	NULL	TIMEZONE_MINUTE
DECLARE	NULLIF	TO
DEFAULT	NUMERIC	TRAILING
DEFERRABLE	OBJECT	TRANSACTION
DEFERRED	OCTET_LENGTH	TRANSLATE
DELETE	OF	TRANSLATION
DEPTH	OFF	TRANSPOSE*
DEREF	OID*	TREAT
DESC	OLD	TRIGGER
DESCRIBE	ON	TRIM

Table B-1. Reserved SQL/MX and SQL/MP Identifiers (page 4 of 4)

DESCRIPTOR	ONLY	TRUE
DESTROY	OPEN	UNDER
DESTRUCTOR	OPERATORS*	UNION
DETERMINISTIC	OPTION	UNIQUE
DIAGNOSTICS	OR	UNKNOWN
DISTINCT	ORDER	UNNEST
DICTIONARY	ORDINALITY	UPDATE
DISCONNECT	OTHERS*	UPPER
DOMAIN	OUT	UPSHIFT*
DOUBLE	OUTER	USAGE
DROP	OUTPUT	USER
DYNAMIC	OVERLAPS	USING
EACH	PAD	VALUE
ELSE	PARAMETER	VALUES
ELSEIF*	PARAMETERS	VARCHAR
END	PARTIAL	VARIABLE
END-EXEC	PENDANT*	VARYING
EQUALS	POSITION	VIEW
ESCAPE	POSTFIX	VIRTUAL*
EXCEPT	PRECISION	VISIBLE*
EXCEPTION	PREFIX	WAIT*
EXEC	PREORDER	WHEN
EXECUTE	PREPARE	WHENEVER
EXISTS	PRESERVE	WHERE
EXTERNAL	PRIMARY	WHILE*
EXTRACT	PRIOR	WITH
FALSE	PRIVATE*	WITHOUT
FETCH	PRIVILEGES	WORK
FIRST	PROCEDURE	WRITE
FLOAT	PROTECTED*	YEAR
		ZONE

SQL/MP Identifiers to Avoid

Words in this list are not reserved in NonStop SQL/MP; however, NonStop SQL/MX considers these words to be reserved in SQL/MX text and in SQL/MP stored text. To prevent problems accessing or manipulating data, avoid using these words as identifiers for SQL/MP objects.

BOTH	DAY	LEADING	SECOND	UNION
CASE	DEFAULT	MINUTE	THEN	WHEN
COLLATE	FRACTION	MONTH	TIME	YEAR
DATE	HOUR	ON	TIMESTAMP	
DATETIME	INTERVAL	RIGHT	TRAILING	

C Limits

This appendix lists limits for various parts of NonStop SQL/MX:

Catalog names	128 characters in length.
Column names	128 characters in length.
Constraints	The maximum combined length of the columns for a REFERENCE, PRIMARY KEY, or UNIQUE constraint depends on the block size. For 4K blocks, the maximum length is 2010 bytes. For 32K blocks, the maximum length is 2048 bytes.
DROP SCHEMA CASCADE transactions	You might need to increase the number of locks per volume, or DROP SCHEMA CASCADE can fail.
EXTENTS	Limited only by size of disk.
FROM clause of the SELECT statement	NonStop SQL/MX typically generates good plans up to 16 tables and tries to generate plans with acceptable performance for queries up to 40 tables. Beyond that limit, the queries require some tuning to compile and run.
IN predicate	1900 expressions are allowed.
Indexes	The maximum combined length of the columns for an index depends on the block size. For 4K blocks, the maximum length is 2010 bytes. For 32K blocks, the maximum length is 2048 bytes. There is no restriction on the number of indexes per table but creating many indexes on a table will affect performance. There are no restrictions on the number of partitions an index supports, but beyond 512, performance and memory issues may occur.
Joins	40 tables can be joined, including base tables of views, but joining more tables affects performance.
MAXEXTENTS size	768 (compare with 919 for NonStop SQL/MP)
Partitions	There are no restrictions on the number of partitions an table can support, but beyond 512, performance and memory issues may occur. Partitions can be on the same physical disk as the main file (SQL/MP partitions must be on a different disk.)
PFS usage	PFS usage is decreased in the file system. This issue affects the number of opens.

Referential constraints	A table can have an unlimited number of referential constraints, and you can specify the same foreign key in more than one referential constraint, but you must define each referential constraint separately.
Schema names	128 characters in length.
Tables	ANSI names are three-part name of the form *catalog.schema.object*, where each part can be up to 128 characters long. The maximum length of a row is 4036 bytes for 4K blocks and 32708 bytes for 32K blocks, but not all of that is available. Depending on data types you use, NonStop SQL/MX might use some bytes for internal use. The clustering key for a table cannot be longer than 2010 bytes for 4K blocks and 2048 bytes for 32K blocks.

D Sample Database

To help you become familiar with its features, NonStop SQL/MX includes a sample database, which you can access by using SQL/MX statements. The sample database is used as the basis for many examples in this manual and other SQL/MX manuals. The DDL statements in the sample database use SQL/MX tables. If you are using SQL/MP tables, you need to use the SQL/MX Release 1.8 sample database, which uses SQL/MP tables.

Note. The SQL/MX Release 2.x sample database uses SQL/MX format tables. To install the sample database, you must have a license to use SQL/MX DDL statements. To acquire this license, purchase product T0394. Without this product, you cannot install the sample database; an error message informs you that the system is not licensed.

This appendix describes:

- Object Names in Sample Database
- Sample Database Entity-Relationship Diagram
- DDL Statements for the Sample Database

For more information on how to install the sample database, see the *SQL/MX Quick Start*.

Object Names in Sample Database

The catalog name for the sample database is `samdbcat` by default, and the catalog contains three schemas—PERSNL, SALES, and INVENT.

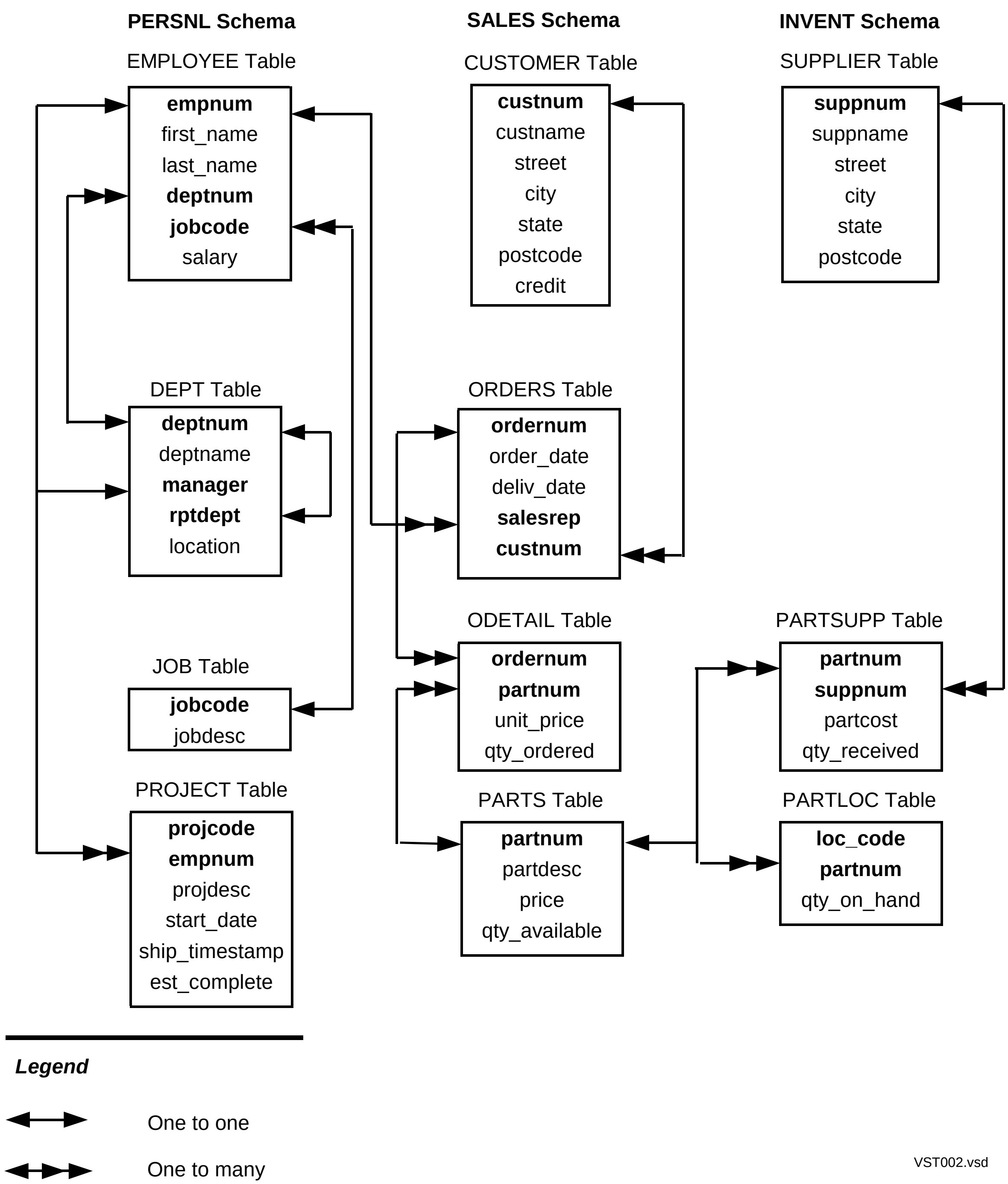
PERSNL	Contains the EMPLOYEE Table, JOB Table, DEPT Table, and PROJECT Table, which are used to store personnel data.
SALES	Contains the CUSTOMER Table, ORDERS Table, ODETAIL Table, and PARTS Table, which are used to store order data.
INVENT	Contains the SUPPLIER Table, PARTSUPP Table, and PARTLOC Table, which are used to store inventory data.

The DML examples in this manual use three-part logical names, *`catalog.schema.name`*, because ANSI SQL:1999 applications use logical names.

Sample Database Entity-Relationship Diagram

Figure D-1 shows the names of columns and tables and the relationships between the tables in the sample database.

Figure D-1. Sample Database Tables



DDL Statements for the Sample Database

The data definition language statements that create sample database objects are listed next. For more information on the `setmxdb` script and how to install the sample database, see the *SQL/MX Quick Start*.

EMPLOYEE Table

This statement creates the EMPLOYEE table:

```
CREATE TABLE samdbcat.persnl.employee
    ( empnum               NUMERIC (4) UNSIGNED
                           NO DEFAULT
                           NOT NULL NOT DROPPABLE
                           HEADING 'Employee/Number'
      ,first_name          CHARACTER (15)
                           DEFAULT ' '
                           NOT NULL NOT DROPPABLE
                           HEADING 'First Name'
      ,last_name           CHARACTER (20)
                           DEFAULT ' '
                           NOT NULL NOT DROPPABLE
                           HEADING 'Last Name'
      ,deptnum             NUMERIC (4)
                           UNSIGNED
                           NO DEFAULT
                           NOT NULL NOT DROPPABLE
                           HEADING 'Dept/Num'
      ,jobcode             NUMERIC (4) UNSIGNED
                           DEFAULT NULL
                           HEADING 'Job/Code'
      ,salary              NUMERIC (8, 2) UNSIGNED
                           DEFAULT NULL
                           HEADING 'Salary'
      ,PRIMARY KEY         (empnum) NOT DROPPABLE
    );
```

This statement creates the EMPNUM_CONSTRNT constraint:

```
ALTER TABLE employee
   ADD CONSTRAINT empnum_constrnt
   CHECK (empnum BETWEEN 0001 and 9999);
```

This statement creates the XEMPNAME index:

```
CREATE INDEX xempname
   ON employee (
                 last_name
               , first_name
                 );
```

This statement creates the XEMPDEPT index:

```
CREATE INDEX xempdept
    ON employee (
```

```
                    deptnum
                      );
```

This statement creates the EMPLIST view:

```
CREATE VIEW emplist
   AS SELECT
        empnum
        , first_name
        , last_name
        , deptnum
        , jobcode
   FROM employee;
```

DEPT Table

This statement creates the DEPT table:

```
CREATE TABLE samdbcat.persnl.dept
    ( deptnum             NUMERIC (4) UNSIGNED
                          NO DEFAULT
                          NOT NULL NOT DROPPABLE
                          HEADING 'Dept/Num'
      ,deptname           CHARACTER (12)
                          NO DEFAULT
                          NOT NULL NOT DROPPABLE
                          HEADING 'Dept/Name'
      ,manager            NUMERIC (4) UNSIGNED
                          NO DEFAULT
                          NOT NULL NOT DROPPABLE
                          HEADING 'Mgr'
      ,rptdept            NUMERIC (4) UNSIGNED
                          DEFAULT 0
                          NOT NULL NOT DROPPABLE
                          HEADING 'Rpt/Dept'
      ,location           VARCHAR (18)
                          DEFAULT 0
                          NOT NULL NOT DROPPABLE
                          HEADING 'Location'
      ,PRIMARY KEY        (deptnum) NOT DROPPABLE
    );
```

This statement creates the MGRNUM_CONSTRNT constraint:

```
ALTER TABLE dept
  ADD CONSTRAINT mgrnum_constrnt
     CHECK (manager BETWEEN 0000 AND 9999);
```

This statement creates the DEPTNUM_CONSTRNT constraint:

```
ALTER TABLE dept
 ADD CONSTRAINT deptnum_constrnt
```

```
    CHECK (deptnum IN (
                         1000
                        ,1500
                        ,2000
                        ,2500
                        ,3000
                        ,3100
                        ,3200
                        ,3300
                        ,3500
                        ,4000
                        ,4100
                        ,9000
                         ));
```

This statement creates the XDEPTMGR index:

```
CREATE INDEX xdeptmgr
  ON dept     (
               manager
               );
```

This statement creates the XDEPTRPT index:

```
CREATE INDEX xdeptrpt
  ON dept   (
             rptdept
             );
```

This statement creates the MGRLIST view:

```
CREATE VIEW mgrlist  (
                       first_name
                      ,last_name
                      ,department
                      )
     AS SELECT
          first_name
         ,last_name
         ,deptname
        FROM
          dept
         ,employee
        WHERE
          dept.manager = employee.empnum;
```

JOB Table

This statement creates the JOB table:

```
CREATE TABLE samdbcat.persnl.job
    ( jobcode            NUMERIC (4) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Job/Code'
```

```
        ,jobdesc            VARCHAR (18)
                            DEFAULT '0'
                            NOT NULL NOT DROPPABLE
                            HEADING 'Job Description'
        ,PRIMARY KEY        (jobcode) NOT DROPPABLE
      );
```

PROJECT Table

This statement creates the PROJECT table:

```
CREATE TABLE samdbcat.persnl.project
    ( projcode             NUMERIC (4) UNSIGNED
                           NO DEFAULT
                           NOT NULL NOT DROPPABLE
                           HEADING 'Project/Code'
      ,empnum              NUMERIC (4) UNSIGNED
                           NO DEFAULT
                           NOT NULL NOT DROPPABLE
                           HEADING 'Employee/Number'
      ,projdesc            VARCHAR (18)
                           DEFAULT '0'
                           NOT NULL NOT DROPPABLE
                           HEADING 'Project/Description'
      ,start_date          DATE
                           DEFAULT DATE '2002-07-01'
                           NOT NULL NOT DROPPABLE
                           HEADING 'Start/Date'
      ,ship_timestamp      TIMESTAMP
                           DEFAULT TIMESTAMP '2002-08-01:12:00:00.000000'
                           NOT NULL NOT DROPPABLE
                           HEADING 'Timestamp/Shipped'
      ,est_complete        INTERVAL DAY
                           DEFAULT INTERVAL '30' DAY
                           NOT NULL NOT DROPPABLE
                           HEADING 'Estimated/Completion'
      ,PRIMARY KEY         (projcode) NOT DROPPABLE
    );
```

CUSTOMER Table

This statement creates the CUSTOMER table:

```
CREATE TABLE samdbcat.sales.customer
      ( custnum             NUMERIC (4) UNSIGNED
                            NO DEFAULT
                            NOT NULL NOT DROPPABLE
                            HEADING 'Cust/Num'
        ,custname           CHARACTER (18)
                            NO DEFAULT
                            NOT NULL NOT DROPPABLE
                            HEADING 'Customer Name'
        ,street             CHARACTER (22)
                            NO DEFAULT
                            NOT NULL NOT DROPPABLE
                            HEADING 'Street
```

```
      ,city                  CHARACTER (14)
                             NO DEFAULT
                             NOT NULL NOT DROPPABLE
                             HEADING 'City'
      ,state                 CHARACTER (12)
                             DEFAULT 0
                             NOT NULL NOT DROPPABLE
                             HEADING 'State'
      ,postcode              CHARACTER (10)
                             NO DEFAULT
                             NOT NULL NOT DROPPABLE
                             HEADING 'Post Code'
      ,credit                CHARACTER (2)
                             DEFAULT 'C1'
                             NOT NULL NOT DROPPABLE
                             HEADING 'CR'
      ,PRIMARY KEY           (custnum) NOT DROPPABLE
    );
```

This statement creates the XCUSTNAM index:

```
CREATE INDEX xcustnam
  ON customer (
                custname
              );
```

This statement creates the CUSTLIST view:

```
CREATE VIEW custlist
     AS SELECT
           custnum
          ,custname
          ,street
          ,city
          ,state
          ,postcode
        FROM customer;
```

ORDERS Table

This statement creates the ORDERS table:

```
CREATE TABLE samdbcat.sales.orders
    ( ordernum               NUMERIC (6) UNSIGNED
                             NO DEFAULT
                             NOT NULL NOT DROPPABLE
                             HEADING 'Order/Num'
      ,order_date            DATE
                             DEFAULT DATE '2002-07-01'
                             NOT NULL NOT DROPPABLE
                             HEADING 'Order/Date'
```

```
    ,deliv_date         DATE
                        DEFAULT DATE '2002-08-01'
                        NOT NULL NOT DROPPABLE
                        HEADING 'Deliv/Date'
    ,salesrep           NUMERIC (4) UNSIGNED
                        DEFAULT 0
                        NOT NULL NOT DROPPABLE
                        HEADING 'Sales/Rep'
    ,custnum            NUMERIC (4) UNSIGNED
                        NO DEFAULT
                        NOT NULL NOT DROPPABLE
                        HEADING 'Cust/Num'
    ,PRIMARY KEY        (ordernum) NOT DROPPABLE
  );
```

DATE_CONSTRNT Constraint

This statement creates the DATE_CONSTRNT constraint:

```
ALTER TABLE orders
  ADD CONSTRAINT date_constrnt
    CHECK (deliv_date >= order_date);
```

This statement creates the XORDREP index:

```
CREATE INDEX xordrep
  ON orders   (
                salesrep
              );
```

This statement creates the XORDCUS index:

```
CREATE INDEX xordcus
  ON orders   (
                custnum
              );
```

This statement creates the ORDREP view:

```
CREATE VIEW ordrep
     AS SELECT empnum
              ,last_name
              ,ordernum
              ,o.custnum
     FROM
        samdbcat.persnl.employee e
       ,samdbcat.sales.orders o
       ,samdbcat.sales.customer c
     WHERE
         e.empnum = o.salesrep
       AND
         o.custnum = C.custnum;
```

ODETAIL Table

This statement creates the ODETAIL table:

```
CREATE TABLE samdbcat.sales.odetail
    ( ordernum           NUMERIC (6) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Order/Num'
      ,partnum           NUMERIC (4) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Part/Num'
      ,unit_price        NUMERIC (8, 2)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Unit/Price'
      ,qty_ordered       NUMERIC (5) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Qty/Ord'
      ,PRIMARY KEY       (ordernum,partnum) NOT DROPPABLE
    );
```

PARTS Table

This statement creates the PARTS table:

```
CREATE TABLE samdbcat.sales.parts
    ( partnum            NUMERIC (4) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Part/Num'
      ,partdesc          CHARACTER (18)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Part Description'
      ,price             NUMERIC (8, 2)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Price'
      ,qty_available     NUMERIC (5)
                         DEFAULT 0
                         NOT NULL NOT DROPPABLE
                         HEADING 'Qty/Avail'
      ,PRIMARY KEY       (partnum) NOT DROPPABLE
    );
```

This statement creates the XPARTDES index:

```
CREATE INDEX xpartdes
  ON parts    (
                partdesc
              );
```

SUPPLIER Table

This statement creates the SUPPLIER table:

```
CREATE TABLE samdbcat.invent.supplier
    ( suppnum            NUMERIC (4) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Supp/Num'
      ,suppname          CHARACTER (18)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Supplier Name'
      ,street            CHARACTER (22)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Street'
      ,city              CHARACTER (14)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'City'
      ,state             CHARACTER (12)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'State'
      ,postcode          CHARACTER (10)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Post Code'
      ,PRIMARY KEY       (suppnum) NOT DROPPABLE
    );
```

This statement creates the XSUPPNAM index:

```
CREATE INDEX xsuppnam
  ON  supplier (
                 suppname
               );
```

PARTSUPP Table

This statement creates the PARTSUPP table:

```
CREATE TABLE samdbcat.invent.partsupp
    ( partnum            NUMERIC (4) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Part/Num'
      ,suppnum           NUMERIC (4) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Supp/Num'
      ,partcost          NUMERIC (8, 2)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Part/Cost'
      ,qty_received      NUMERIC (5) UNSIGNED
                         DEFAULT 0
                         NOT NULL NOT DROPPABLE
                         HEADING 'Qty/Rec'
      ,PRIMARY KEY       (partnum,suppnum) NOT DROPPABLE
    );
```

This statement creates the XSUPORD index:

```
CREATE INDEX xsupord
  ON partsupp (
                suppnum
              );
```

This statement creates the VIEW207 view:

```
CREATE VIEW view207  (
                        partnumber
                       ,partdescrpt
                       ,suppnumber
                       ,supplrname
                       ,partprice
                       ,qtyreceived
                     )
     AS SELECT
           x.partnum
          ,partdesc
          ,x.suppnum
          ,suppname
          ,partcost
          ,qty_received
        FROM
           samdbcat.invent.partsupp x
          ,samdbcat.sales.parts p
          ,samdbcat.invent.supplier s
        WHERE
```

```
        x.partnum  = p.partnum
      AND
        x.suppnum = s.suppnum;
```

This statement creates the VIEW207N view:

```
CREATE VIEW view207n (
                        partnumber
                       ,partdescrpt
                       ,suppnumber
                       ,supplrname
                       ,partprice
                       ,qtyreceived
                     )
     AS SELECT
          x.partnum
         ,p.partdesc
         ,s.suppnum
         ,s.suppname
         ,x.partcost
         ,x.qty_received
        FROM
          samdbcat.invent.supplier s LEFT JOIN
          samdbcat.invent.partsupp x
                ON s.suppnum = x.suppnum
                   LEFT JOIN samdbcat.sales.parts p
                        ON x.partnum = p.partnum;
```

This statement creates the VIEWCUST view:

```
CREATE VIEW viewcust (
                        custnumber
                       ,cusname
                       ,ordernum
                     )
     AS SELECT
          c.custnum
         ,c.custname
         ,o.ordernum
        FROM
          samdbcat.sales.customer c LEFT JOIN
          samdbcat.salesorders o
                 ON c.custnum = o.custnum;
```

This statement creates the VIEWCS view:

```
CREATE VIEW samdbcat.invent.viewcs
     AS SELECT
          custname
        FROM samdbcat.sales.customer
      UNION
        SELECT
          suppname
        FROM samdbcat.invent.supplier;
```

PARTLOC Table

When you install the sample database, you can choose to partition the PARTLOC. If you do not specify partitions when you run `setmxdb`, the unpartitioned PARTLOC table is created:

```
CREATE TABLE samdbcat.invent.partloc
    ( loc_code           CHARACTER (3)
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Loc/Code'
      ,partnum           NUMERIC (4) UNSIGNED
                         NO DEFAULT
                         NOT NULL NOT DROPPABLE
                         HEADING 'Part/Num'
      ,qty_on_hand       NUMERIC (7)
                         DEFAULT 0
                         NOT NULL NOT DROPPABLE
                         HEADING 'Qty/On/Hand'
      ,PRIMARY KEY       (loc_code,partnum) NOT DROPPABLE
    );
```

If you specify partitions when you run `setmxdb`, this statement creates the partitioned PARTLOC table:

```
CREATE TABLE samdbcat.invent.partloc
     ( loc_code          CHARACTER (3)
                         NO DEFAULT
                         NOT NULL
                         HEADING 'Loc/Code'
       ,partnum          NUMERIC (4) UNSIGNED
                         NO DEFAULT
                         NOT NULL
                         HEADING 'Part/Num'
       ,qty_on_hand      NUMERIC (7)
                         DEFAULT 0
                         HEADING 'Qty/On/Hand'
       ,PRIMARY KEY     (loc_code,partnum))
                         LOCATION $PART1VOL
                         PARTITION (ADD FIRST KEY 'P00'
                         LOCATION $PART2VOL
     );
```

E Standard SQL and SQL/MX

This appendix describes NonStop SQL/MX conformance to the SQL standards established by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). It describes:

This appendix documents NonStop SQL/MX conformance with the standards criteria for SQL:1999, which replaced ANSI SQL-92. The mandatory portion of SQL:1999 is referred to as Core SQL:1999 and is found in SQL:1999 Part 2 (Foundation) and Part 5 (Bindings). Core SQL:1999 contains all Entry SQL-92, much of Intermediate SQL-92, and some of Full SQL-92, including some new SQL:1999 features.

Annex F of Part 2 in the table “SQL/Foundation feature taxonomy and definition for Core SQL” describes Foundation features. Annex F of Part 5 in the table “SQL/Bindings feature taxonomy and definition for Core SQL” describes Bindings features.

ANSI SQL Standards

These ANSI documents govern SQL:

- ANSI/ISO/IEC 9075-1:1999, Information technology—Database languages—SQL—Part 1: Framework (SQL/Framework)
- ANSI/ISO/IEC 9075-1:1999/Amd.1:2000
- ANSI/ISO/IEC 9075-2:1999, Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)
- ANSI/ISO/IEC 9075-5:1999, Information technology—Database languages—SQL—Part 5: Host Language Bindings (SQL/Bindings)

To obtain copies of the ANSI standards, refer to the ANSI eStandards Store at:

http://webstore.ansi.org/ansidocstore/default.asp

The X3 or INCITS standards offer a subset of the ANSI standards that include the SQL standard. To obtain copies, see the International Committee for Information Technology Standards at:

http://www.cssinfo.com/ncitsgate.html

ISO Standards

These ISO documents govern SQL:

- ISO/IEC 9075-1:1999, Information technology—Database languages—SQL—Part 1: Framework (SQL/Framework)
- ISO/IEC 9075-1:1999/Amd.1:2000
- ISO/IEC 9075-2:1999, Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)
- ISO/IEC 9075-5:1999, Information technology—Database languages—SQL—Part 5: Host Language Bindings (SQL/Bindings)

For more information about ISO standards, see:

- ISO Web site: http://www.iso.ch/
- ISO Store Web site: http://www.iso.org/iso/en/prods-services/ISOstore/store.html

SQL/MX Compliance

The ANSI and ISO SQL standards require conformance claims to state the type of conformance and the implemented facilities. SQL/MX products provide full or partial conformance. This table lists the Core SQL:1999 features for which NonStop SQL/MX offers full conformance:

ID	Feature
E011	Numeric data types
E021	Character data types
E031	Identifiers
E051	Basic query specification
E061	Basic predicates and search conditions
E081	Basic privileges
E091	Set functions
E101	Basic data manipulation
E111	Single row SELECT statement
E131	Null value support (nulls in lieu of values)
E151	Transaction support
E152	Basic SET TRANSACTION statement
E171	SQLSTATE support
F031	Basic schema manipulation
F041	Basic joined table
F131	Grouped operations

ID	Feature
F181	Multiple module support
F201	CAST function
F471	Scalar subquery values
F481	Expanded NULL predicate
B012	Embedded C
B013	Embedded COBOL

This table lists the Core SQL:1999 features for which NonStop SQL/MX offers partial support:

ID, Feature	Level of Support
E071 Basic query expressions	NonStop SQL/MX fully supports this subfeature: E071-02 UNION ALL table operator NonStop SQL/MX partially supports these subfeatures: E071-01 UNION [DISTINCT] table operator (NonStop SQL/MX does not support explicitly specifying DISTINCT) E071-05 Columns combined by table operators need not have exactly the same type (NonStop SQL/MX has rules for determining result types that are not identical to SQL '99 rules) E071-06 table operators in subqueries (NonStop SQL/MX allows only UNION in subqueries) NonStop SQL/MX does not support this subfeature: E071-03 EXCEPT [DISTINCT] table operator
E121 Basic cursor support	NonStop SQL/MX fully supports these subfeatures: E121-01 DECLARE CURSOR E121-02 ORDER BY columns need not be in select list E121-06 Positioned UPDATE statement E121-07 Positioned DELETE statement E121-08 CLOSE statement NonStop SQL/MX partially supports these subfeatures: E121-04 OPEN statement (SQL/MX syntax does not match that of SQL '99) E121-10 FETCH statement, implicit NEXT (SQL/MX syntax does not match that of SQL '99) E121-17 WITH HOLD cursors (supported only for SELECT statements that use stream access mode or an embedded UPDATE or embedded DELETE)

ID, Feature	Level of Support
	NonStop SQL/MX does not support this subfeature: E121-03 Value expressions in ORDER BY clause
E141 Basic integrity constraints	NonStop SQL/MX fully supports these subfeatures: E141-01 NOT NULL constraints E141-02 UNIQUE constraint of NOT NULL columns E141-03 PRIMARY KEY constraints E141-06 CHECK constraints NonStop SQL/MX partially supports these subfeatures: E141-04 Basic FOREIGN KEY constraint with the NO ACTION default for referential delete and update action(s) (NonStop SQL/MX has the limitation that the “referenced-table cannot be the same as table”) E141-07 Column defaults (NonStop SQL/MX specifies a subset of `datetime value function`s that Core '99 allows to be specified. NonStop SQL/MX does not enforce the conformance rule that “Without Feature F411 “Time zone specification”, CURRENT_TIME and CURRENT_TIMESTAMP shall not be specified.”) NonStop SQL/MX does not support these subfeatures: E141-08 NOT NULL inferred on PRIMARY KEY (NOT NULL is not required to be inferred on UNIQUE constraints) E141-10 Names in a foreign key can be specified in any order (The column in the column list associated with REFERENCES must be in the same order as the column in the column list associated with FOREIGN KEY)
E161 SQL comments using leading double minus	MXCI does not properly process SQL comments when they are issued from the NonStop Manager
F051 Basic date and time	NonStop SQL/MX fully supports these subfeatures: F051-01 DATE data type (and literal) F051-02 TIME data type (and literal) with fractional seconds precision of at least 0 F051-03 TIMESTAMP data type (and literal) with fractional seconds precision of at least 0 and 6 F051-04 Comparison predicate for DATE, TIME, and TIME-STAMP data types F051-05 Explicit CAST between datetime types and character types F051-06 CURRENT_DATE

ID, Feature	Level of Support
	NonStop SQL/MX does not support these subfeatures: F051-07 LOCALTIME (equivalent to CAST (CURRENT_TIME AS TIME WITHOUT TIME ZONE)) F051-08 LOCALTIMESTAMP (equivalent to CAST(CURRENT_TIMESTAMP AS TIMESTAMP WITHOUT TIME ZONE))
F081 UNION and EXCEPT in views	NonStop SQL/MX supports UNION but not EXCEPT in views
F221 Explicit defaults	NonStop SQL/MX supports use of DEFAULT in INSERT, but not in UPDATE
F261 CASE expressions	NonStop SQL/MX supports these subfeatures: F261-01 Simple CASE F261-02 Searched CASE NonStop SQL/MX does not support these subfeatures: F261-03 NULLIF F261-04 COALESCE
F311 Schema definition statement	NonStop SQL/MX partially supports these subfeatures: F311-01 CREATE SCHEMA (NonStop SQL/MX does not support creation of a schema and its contents in a single statement. Objects must be created in a particular order, dependent objects first.) F311-02 CREATE TABLE for persistent base tables (NonStop SQL/MX does not support creation of a schema and its contents in a single statement. Objects must be created in a particular order, dependent objects first.) F311-03 CREATE VIEW (without WITH CHECK OPTION and without Feature F081 “UNION and EXCEPT in views” SQL/MX views cannot refer to tables created in the same CREATE SCHEMA) F311-04 CREATE VIEW: WITH CHECK OPTION (Without support for Feature F081 “UNION and EXCEPT in views” SQL/MX views cannot refer to tables created in the same CREATE SCHEMA) F311-05 GRANT statement (NonStop SQL/MX does not support creation of a schema and its contents in a single statement, including performing grant operations. Objects must be created in a particular order, dependent objects first.)
T321 Basic SQL-invoked routines	NonStop SQL/MX fully supports this subfeature: T321-04 CALL statement

NonStop SQL/MX does not support these Core SQL:1999 features:

ID	Feature
E153	Updatable queries with subqueries
F021	Basic information schema
F501	Features and conformance views
F812	Basic flagging
S011	Distinct data types

NonStop SQL/MX supports embedded language. NonStop SQL/MX does not support E182, module language. Even though this feature is listed in Table 31 in the standard, Core SQL:1999 allows a choice between module language and embedded language. Module language and embedded language are identical in capability. They differ only in how SQL statements are associated with the host programming language.

SQL/MX Extensions to Standard SQL

NonStop SQL/MX provides many features that enhance or supplement the functionality of standard SQL. In your SQL/MX applications, you can use these extensions just as you can use Core SQL:1999. This table shows the Non-Core extensions that NonStop SQL/MX supports:

ID	Feature
B021	Direct SQL
B031	Basic dynamic SQL
B032	Extended dynamic SQL
F111	Isolation levels other than SERIALIZABLE
F121	Basic diagnostics management
F171	Multiple schemas per user
F222	INSERT statement: DEFAULT VALUES clause
F281	LIKE enhancements
F321	User authorization (no support for SYSTEM_USER)
F381-02	ALTER TABLE statement: ADD CONSTRAINT clause
F381-03	ALTER TABLE statement: DROP CONSTRAINT clause
F401-01	NATURAL JOIN
F401-04	CROSS JOIN
F461	Named character sets
F491	Constraint management
F561	Full value expressions
F651	Catalog name qualifiers
T171	LIKE clause in table definition (not exact Core SQL:1999 syntax)

ID	Feature
T211	Basic trigger capability (except for T211-07 Trigger privilege)
T212	Enhanced trigger capability
T441	ABS and MOD functions
T621	Enhanced numeric functions

Character Set Support

NonStop SQL/MX supports a limited number of national, international, and vendor-specific encoded character set standards: ISO88591, UCS2, KANJI and KSC5601.

Note. KANJI and KSC5601 are valid character sets for SQL/MP tables but not SQL/MX tables. If you attempt to create an SQL/MX table with KANJI, KSC5601, or other unsupported character sets, you get an SQL error and the operation fails.

Unicode is a universal encoded character set that lets you store information from any language using a single character set. Modern standards, such as XML, Java, Java Script, and LDAP, require Unicode. Unicode complies with ISO/IEC standard 10646. To obtain a copy of ISO/IEC standard 10646, see the Web sites listed under ISO Standards on page E-2.

SQL/MX Release 2.x complies fully with the Unicode 2.1 standard. For more information about this standard, see the Web site of the Unicode Consortium:

http://www.unicode.org

NonStop SQL/MX uses UTF-16BE (16-bit) encoding for the Unicode (UCS2) character set. The full range of UTF-16 characters is allowed, but surrogate pairs are not recognized as characters. NonStop SQL/MX uses single-byte encoding for ISO88591 character set and permits wide-character KANJI and KSC5601 character set host variables in embedded applications that query SQL/MP tables.

NonStop SQL/MX relaxes SQL:1999's data type matching rule for UCS2 character set host variables for ease of use and better performance. A UCS2 host variable is assignment and comparison compatible with an ISO88591 value expression.

NonStop SQL/MX allows various SQL:1999's NATIONAL CHARACTER syntax to denote a predesignated character set. The default NATIONAL character set is UCS2. You can specify a different default character set during SQL/MX installation.

NonStop SQL/MX uses binary collation (that is, comparison of binary code values) to compare character strings.

NonStop SQL/MX complies fully with SQL:1999 for these character data type subfeatures: character string data type declaration, character value expression, search condition, string functions and predicates, string literals and host variables in C/COBOL embedded programs.

Index

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Special Characters