HP NonStop SQL/MP Installation and Management Guide

Abstract

This manual explains how to install HP NonStop™ SQL/MP, the HP relational database management system, and how to plan, create, and manage SQL/MP databases and applications.

Product Version

NonStop SQL/MP G06

Supported Release Version Updates (RVUs)

This publication supports G06.22 and all subsequent G-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
523353-004	December 2004

Document History

Part Number	Product Version	Published
520680-001	NonStop SQL/MP G06	November 2001
523353-001	NonStop SQL/MP G06	February 2002
523353-002	NonStop SQL/MP G06	December 2003
523353-003	NonStop SQL/MP G06	March 2004
523353-004	NonStop SQL/MP G06	December 2004

HP NonStop SQL/MP Installation and Management Guide

Index Figures Tables

4. Planning Database Security and Recovery (continued)

7. Adding, Altering, Removing, and Renaming Database Objects (continued)

8. Reorganizing Tables and Maintaining Data (continued)

9. Moving a Database

10. Managing Database Applications

11. Performing Recovery Operations

11. Performing Recovery Operations (continued)

14. Enhancing Performance (continued)

What’s New in This Manual

Manual Information

Abstract

This manual explains how to install HP NonStop™ SQL/MP, the HP relational database management system, and how to plan, create, and manage SQL/MP databases and applications.

Product Version

NonStop SQL/MP G06

Supported Release Version Updates (RVUs)

This publication supports G06.22 and all subsequent G-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
523353-004	December 2004

Document History

Part Number	Product Version	Published
520680-001	NonStop SQL/MP G06	November 2001
523353-001	NonStop SQL/MP G06	February 2002
523353-002	NonStop SQL/MP G06	December 2003
523353-003	NonStop SQL/MP G06	March 2004
523353-004	NonStop SQL/MP G06	December 2004

New and Changed Information

- Updated the examples for the INSERT statement on page 9-13 and page 9-14.
- Added new information under PATHMON DEFINEs and SQL Recompilation on page 10-10.

About This Manual

The SQL/MP relational database management system (RDBMS) uses the Structured Query Language (SQL) to create databases and to describe and manipulate data. SQL/MP is compatible with the American National Standards Institute (ANSI) and International Organization for Standardization (ISO) standards for SQL. SQL/MP provides high performance for production of online transaction processing (OLTP) applications over a full range of centralized or distributed systems.

Audience and Task Analysis

This manual provides comprehensive descriptions of how to perform the tasks of planning, creating, and managing an SQL/MP database, frequently giving step-by-step instructions. As a database administrator or manager, you should find this manual helpful when performing these tasks:

- Installing SQL/MP
- Determining the database layout and data dictionary plan
- Planning for database security, integrity, and recovery
- Creating the database
- Querying catalogs for information about the database
- Managing the database and programs
- Reorganizing the database
- Moving the database
- Managing database applications
- Performing recovery operations
- Managing a distributed database
- Measuring and enhancing performance

You should be familiar with the HP NonStop operating system, the HP NonStop Transaction Management Facility (TMF), and any specific application environments you manage, such as the Pathway transaction processing environment.

Prerequisites

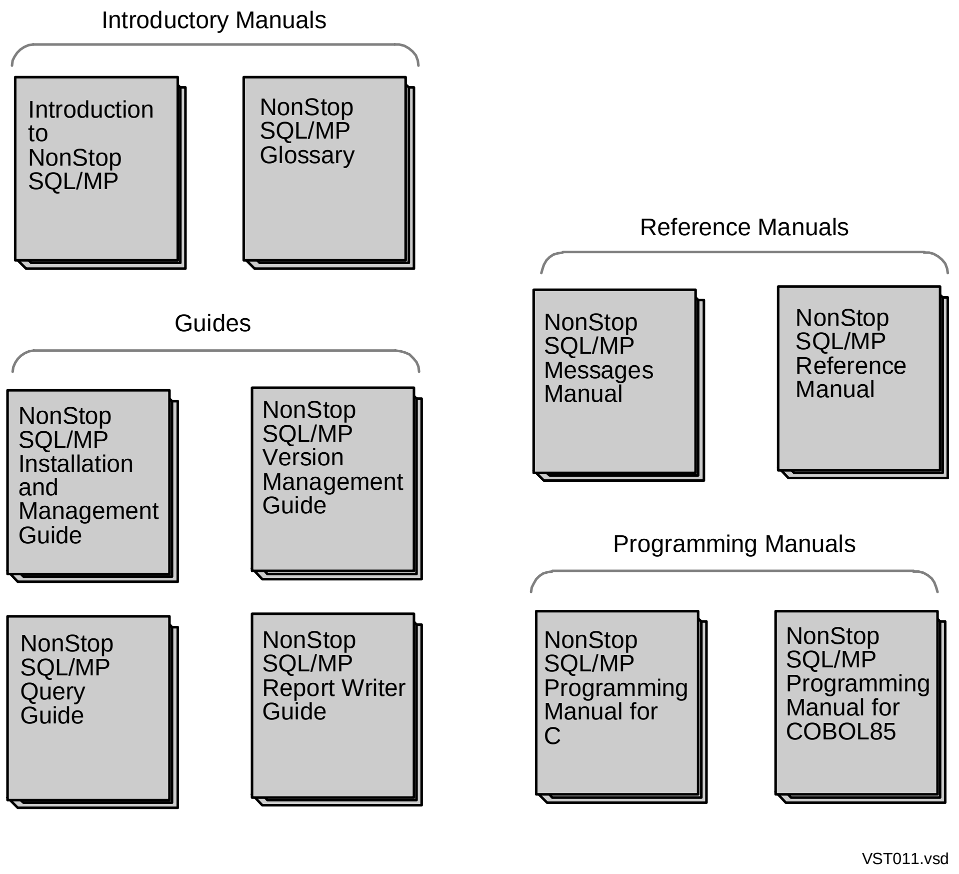
Before using this manual, you should read the *Introduction to NonStop SQL/MP* so that you are familiar with SQL/MP terminology and concepts. You will need a copy of the *SQL/MP Reference Manual*, which documents SQL statements and SQLCI command syntax and provides detailed information about how SQL/MP works. You will also need a copy of the *SQL/MP Query Guide,* which explains how to design and tune database queries, and the *SQL/MP Version Management Guide*, which explains how to work with different versions of SQL/MP software, objects, and programs.

NonStop SQL/MP Library

This manual is a part of the SQL/MP library of manuals. The library also includes these manuals:

- *Introduction to NonStop SQL/MP* provides an overview of the SQL/MP relational database management system.
- *SQL/MP Reference Manual* describes the syntax and provides examples and usage considerations for: SQL language elements, expressions, functions, and statements; the SQLCI standard conversational commands, SQL utility commands, and session attribute commands; and the SQL/MP report writer commands. This manual is the printed version of SQL/MP Online Help.
- *SQL/MP Version Management Guide* describes the rules governing version management for the SQL/MP software, catalogs, objects, messages, programs, and data structures.
- *SQL/MP Query Guide* describes how to write SQL/MP queries and how to optimize queries for enhanced performance.
- *SQL/MP Report Writer Guide* describes how to use report writer commands and SQLCI options to design and produce reports.
- *SQL/MP Programming Manual* (available for C or COBOL) describes the programmatic interface for the particular host language.
- *SQL/MP Messages Manual* describes the messages issued by SQL/MP software, as well as file-system and FastSort messages returned by SQL/MP.
- *SQL/MP Glossary* defines the terms and expressions used in SQL/MP.

This figure illustrates the relationships between the *SQL/MP Installation and Management Guide* and other manuals in the SQL/MP library.


Introductory Manuals
Introduction to NonStop SQL/MP
NonStop SQL/MP Glossary
Reference Manuals
NonStop SQL/MP Messages Manual
NonStop SQL/MP Reference Manual
Guides
NonStop SQL/MP Installation and Management Guide
NonStop SQL/MP Version Management Guide
Programming Manuals
NonStop SQL/MP Query Guide
NonStop SQL/MP Report Writer Guide
NonStop SQL/MP Programming Manual for C
NonStop SQL/MP Programming Manual for COBOL85
VST011.vsd

Related Manuals

These manuals contain more detailed information about NonStop systems and about software products used with SQL/MP:

- *DataLoader/MP Reference Manual* describes how to use the DataLoader/MP product to load and maintain SQL/MP or Enscribe databases.
- *Guardian Disk and Tape Utilities Reference Manual* describes the BACKUP and RESTORE utilities for backing up files onto tape and restoring them to disk, the Disk Compression Program (DCOM) for moving files to gain more usable disk space, and the Disk Space Analysis Program (DSAP) for analyzing the use of disk space on a volume.
- *FastSort Manual* describes FastSort, the sort-merge product for NonStop systems.
- *File Utility Program (FUP) Reference Manual* describes the utility program for managing files on NonStop systems.

- *Measure Reference Manual* describes the use of the Measure product to collect statistical information on database objects and processes, and to generate reports for performance analysis.
- *Open System Services Shell and Utilities Reference Manual* describes how to use OSS utilities; this information is useful for OSS SQL programs.
- *Peripheral Utility Program (PUP) Reference Manual* contains syntax, considerations, and examples for PUP interactive commands.
- *Safeguard User's Guide* explains how to use the Safeguard security product.
- *Security Management Guide* describes Guardian security in detail and provides examples of authorization schemes.
- *Storage Management Foundation User's Guide* introduces the concepts and components of the NonStop Storage Management Foundation (SMF) product. It contains information on how to migrate an existing system to SMF and how to configure and maintain SMF processes.
- *SCF Reference Manual* describes the operation of SCF on HP NonStop S-series servers and how it is used to configure, control, and inquire about supported systems.
- *TACL Reference Manual* presents the syntax and operations of the standard commands and functions available in the operating system's command interpreter.
- *TMF Planning and Configuration Guide* explains how to configure the TMF subsystem.
- *TMF Operations and Recovery Guide* describes how to use the TMF subsystem to protect a database against disk, system, and program failures.

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken 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

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type 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. Type 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 typed 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 type 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, no spaces are 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 ]…
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                  !i
                         , error        ) ;            !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                    !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length          !i:i
                            , filename2:length ) ;      !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                         !i
                        , [ filename:maxlen ] ) ;         !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE

?123

CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. 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:

```
proc-name trapped [ in SQL | in SQL file system ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. 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:

```
obj-type obj-name state changed to state, caused by
{ Object | Operator | Service }
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

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

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400
```

```
%B101111
```

```
%H2F
```

```
P=%p-register E=%e-register
```

Notation for Management Programming Interfaces

This list summarizes the notation conventions used in the boxed descriptions of programmatic commands, event messages, and error lists in this manual.

UPPERCASE LETTERS. Uppercase letters indicate names from definition files. Type these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters. Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r. The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME      token-type ZSPI-TYP-STRING.          !r
```

!o. The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER      token-type ZSPI-TYP-FNAME32.         !o
```

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 COBOL 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.

1 The SQL/MP Database Management Environment

Managing an SQL/MP relational database typically involves managing sets of continuously active programs in addition to the database itself. In this environment, you must add new applications and disk volumes, and repair or change existing applications, all without affecting other applications currently running on your system.

In this manual, the descriptions of managing a database are based on these assumptions about the database environment at your site:

- An application includes database components (tables, indexes, views, and collations) and the programs that access the database. Database definitions, configuration, and distribution can significantly affect application performance. An SQL/MP database is an integral part of an application.
- The database is a production database, not a test database. The database must be consistent, accurate, and available.
- Application development is ongoing. New or changed applications must be integrated with existing applications.
- Central database management is required. The database management tasks can be performed by groups of people, but these tasks should be administered by a single person or group.

The database management environment for SQL/MP includes:

- The SQL/MP relational database management system (RDBMS)
- Database management tools
    - SQL statements, SQL utilities, and SQL conversational interface (SQLCI) commands
    - Guardian and OSS utilities
    - HP products for database security, conversion, and reorganization

SQL/MP Software

An SQL/MP database is not just a storage mechanism for data. The database is an active part of an application and should be designed for the best application performance. You can design a database for each application, or you can use views to customize a database for more than one application.

Each SQL database consists of a collection of SQL objects, the data dictionary for these objects, and any files in which the objects are stored. The data dictionary includes all catalogs and associated file labels that describe the objects. A catalog can be part of more than one database. Each node (system) can have more than one SQL database, and a database can span more than one node.

You use SQL/MP through SQL statements, SQLCI commands, and SQL compiler directives.

SQL Objects

An SQL/MP database includes these objects:

- One or more base tables
- Indexes on tables
- Views on tables
- Constraints on tables
- Collations used by tables, indexes, and views

Base Tables

A base table is a table of one or more columns for which rows of data are stored in a file. The base table definition specifies the physical characteristics of the table for the file system. Column definitions for the table specify data types and character sets for row values.

Data is entered into the table, one row at a time. Each row contains a value for each table column. Stored data is retrieved and displayed in columns and rows.

The order in which rows are stored in a file is determined by the table organization and the table's primary key. The table organization can be key-sequenced, entry-sequenced, or relative, and the file organization corresponds to the table organization explained in Understanding SQL File Structures on page 3-1. Primary keys are defined as follows:

- The primary key of a table stored in a key-sequenced file is one or more columns defined by the user, and the SQL/MP software, or solely by the SQL/MP software.
- The primary key of a table stored in a relative file is a relative record number defined by the user or the SQL/MP software.
- The primary key for a table stored in an entry-sequenced file is a record address defined by the SQL/MP software.

Indexes

An index is an alternate access path to a table (alternate key) that differs from the primary access path (primary key). An SQL/MP index, stored in a file, includes columns for the table's primary key and the alternate key.

Views

A view is a table that has a logical definition and a file label, but contains no data. A protection view is derived from a single table by selecting a subset of the table columns and rows. A shorthand view is derived by selecting columns and rows from one or more tables and views.

Constraints

A constraint helps protect the integrity of data in a base table by specifying a condition or conditions that all the values in a particular column of the table must satisfy. Unlike other SQL objects, constraints have only SQL names, not Guardian names, and constraints do not have external file labels.

Collations

A collation contains rules for collating sequence, upshifting, downshifting, character class, and character string equivalence. A collation associated with a table, index, or view column defines the default sort order for column values and shifting rules.

SQL Catalogs

An SQL/MP catalog consists of a set of tables on a single subvolume. Catalog tables contain specific information about SQL objects and programs, such as keys, comments, columns, partitions, attributes, and interdependencies. The catalog tables themselves compose a normalized SQL database indexed for rapid access.

The SQL/MP catalog manager process automatically updates one or more sets of catalog tables according to instructions in DDL statements. When you create an SQL object, SQL/MP registers the object in an existing catalog. You can specify which catalog is to contain the description of an object.

Each node that uses SQL/MP has a catalog called the system catalog, that contains information about all the catalogs on the node. The system catalog is like any other catalog, with a few exceptions (described in Requirements for Catalogs on page 5-4).

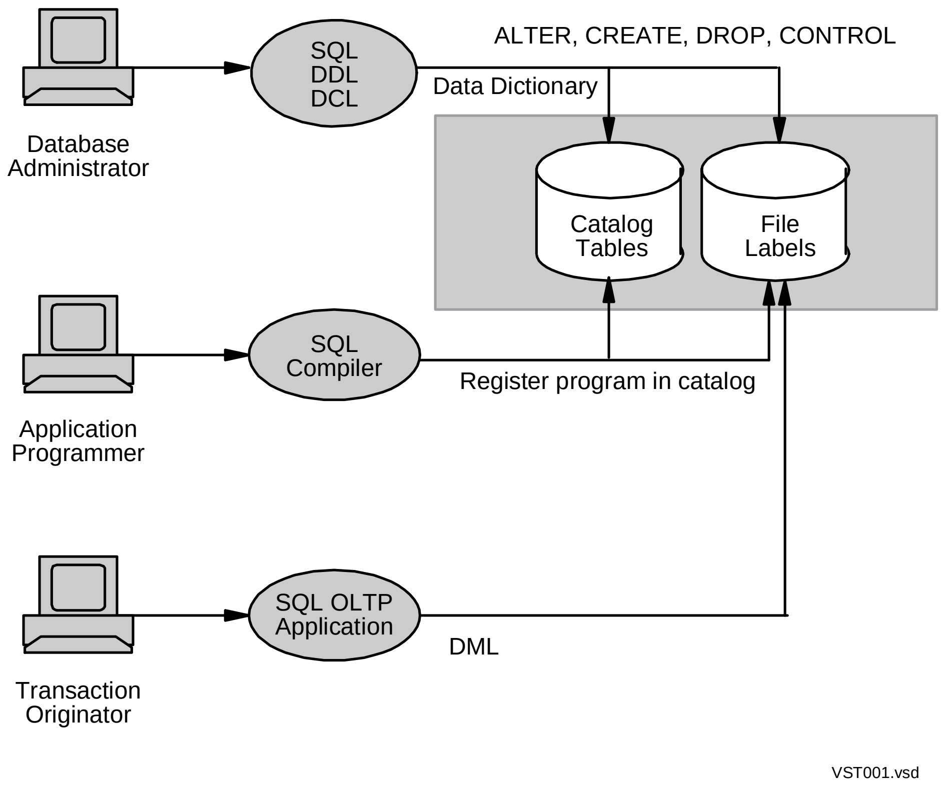
Active Data Dictionary

An SQL/MP data dictionary is the collection of all the catalogs and associated file labels that describe all the SQL/MP tables, views, indexes, constraints, and collations that make up a database.

The SQL/MP data dictionary is an active dictionary. Any changes to database definitions immediately apply to programs that use the changed portion of the database. The data dictionary is not a passive collection of definitions but an active part of applications. When performing database management tasks, you must be aware of how changes affect the data dictionary.

Figure 1-1 shows typical use of a data dictionary. DDL statements and the SQL compiler access both the catalogs and the file labels, but SQL compiled programs access only the file labels of objects during program execution. An executing program requires catalog access only when the program requires automatic recompilation.

Figure 1-1. Typical Use of the Data Dictionary



To understand how the active dictionary affects programs, consider the effect of adding a constraint on a table. Suppose that your company has reorganized the boundaries of its sales divisions, and your division no longer includes the state of New York. As a result of the reorganization, all your New York customers have been given to the other division and have been removed from your database.

The database includes a CUSTOMER table with a STATE column. Now you can add a constraint to prohibit adding any customers from New York. A constraint specifies a rule that rows in a table must satisfy. The statement to create the constraint follows:

```
CREATE CONSTRAINT CUSTOMER_NEW_YORK
   ON CUSTOMER CHECK STATE <> 'NEW YORK'
```

After this constraint is entered, either through SQLCI or through an application program, no user can add a customer from the state of New York. SQL/MP enforces this constraint. No programmatic changes are necessary to protect the database against adding an invalid customer.

The RDBMS ensures that every program uses the current table definition by requiring SQL recompilation of every program that uses the CUSTOMER table. When you do not recompile a program explicitly, an SQL recompilation occurs automatically before each execution of the program to ensure it uses the current definition of the table.

SQL/MP Features

The SQL/MP RDBMS supports:

- Distributed databases
- Database protection and recovery
- Data integrity
- Multiple character sets, including single-byte and double-byte data
- DEFINEs
- Database security
- Parallel processing
- High availability

Distributed Databases

You can distribute an SQL/MP database across multiple nodes with complete transparency, location independence, read and update capability, and data integrity. Objects in the same database can reside on different nodes. The partitions of a table or index can be spread across nodes as well, with the partitioning invisible to users.

Local autonomy guarantees that local processing is not affected by unreliable communications links. This autonomy is facilitated by an active, distributed data dictionary and a feature that opens partitions of a database only when demanded by a query. Also, automatic recompilation is invoked to rebind plans when certain resources are unavailable.

A distributed database can be managed by a central site, by each independent distributed site, or by a combination of these. For more information, see Section 12, Managing a Distributed Database.

Database Protection and Recovery

An SQL/MP database is protected by the TMF subsystem, which provides automatic online recovery of audited SQL objects and files by using audit trails. A transaction in progress can be aborted when a failure or error condition is detected. The TMF subsystem terminates the transaction and restores the database to its state before the beginning of the transaction. The TMF subsystem provides two additional recovery mechanisms:

- Volume recovery, which recovers the database in the event of a disk crash or system failure

- File recovery, which reconstructs specific audited files when the current copies on the data volume are not usable; for example, if a system or media failure jeopardizes the consistency of one or more audited files

The BACKUP utility provides volume-mode and file-mode tape backups for database objects and for SQL programs stored in Guardian files, which you can recover with RESTORE. Volume-mode backup makes a physical copy of a disk volume on tape. File-mode backup copies each SQL object or file in a file set list to tape.

You can also save database creation and loading scripts in OBEY command files or write an automated program to handle TMF and backup and restore operations.

OSS utilities provide backup functionality for SQL programs stored in OSS files.

For more information about database protection and recovery, see The TMF Subsystem on page 4-10 and Restoring Objects With TMF Recovery Operations on page 11-11.

Data Integrity

The database management system protects the database by ensuring that entered data meets the definitional requirements. Application programs, therefore, do not need to perform data checking.

These data definition features ensure definitional integrity:

- Column definitions
- Protection views
- Constraints
- Indexes

These features provide additional data integrity for SQL/MP databases:

- Database changes are monitored by the TMF subsystem.
- Database access is restricted to SQL statements.
- Integrity constraints on tables are enforced by the disk process.
- Database consistency is maintained by concurrency control.

The TMF subsystem provides data integrity. Database updates performed as part of a TMF transaction are either all committed to the database when the transaction completes or all backed out if the transaction does not complete. With the TMF two-phase commit protocol, an update changes a database from one consistent state to another; an incomplete transaction does not change the database. TMF transactions can be distributed across multiple nodes.

Database access is restricted to standard Data Manipulation Language (DML), Data Definition Language (DDL), Data Control Language (DCL), and Data Status Language (DSL) statements. No exit routines can inadvertently corrupt a database.

For each table, you can define one or more integrity constraints that the disk process checks before inserting a row into the table. Each constraint is an SQL object. If a row does not satisfy a condition specified in a constraint, the disk process does not insert the row. Likewise, update values for existing rows must satisfy all constraints on the table.

Concurrency control for database access provides different degrees of database consistency to meet different needs: browse access, stable access, and repeatable access. These access modes are maintained by shared or exclusive locks on rows, sets of rows, partitions, and tables.

Multiple Character Sets

SQL/MP supports multiple character sets with the CHARACTER, PIC X, and VARCHAR data types and the CHARACTER SET clause. In addition, the NATIONAL CHARACTER data type supports double-byte character sets.

DEFINEs

A DEFINE is a named set of attributes and associated values stored in the process file segment (PFS) of a running process. You simplify the SQL/MP environment by using logical DEFINE names in commands, in OBEY command files, and in programs. You can have the names apply to other SQL objects just by changing the DEFINEs.

You can use a DEFINE name to specify a subvolume, catalog, table, view, or index in SQL/MP statements and commands and in host application programs. Typically, you use a DEFINE to establish a one-to-one mapping between a logical name and a physical name. For more information, see Using DEFINEs on page 10-30.

Database Security

Authorization to operate on SQL/MP tables, indexes, views, collations, and SQL programs that run in the Guardian environment is maintained by Guardian security and checked by SQL/MP. You can use the Safeguard security management facility for additional security protection for volumes and subvolumes containing SQL objects and for individual SQL programs stored in Guardian files. For more information, see Security Guidelines on page 4-2.

Authorization for SQL programs stored in the OSS environment is maintained by the OSS security subsystem.

Parallel Processing

The following types of parallel processing are available in SQL/MP:

- Parallel query processing
- Parallel join operations
- Parallel index maintenance
- Parallel index loading
- Parallel sorting
- Parallel input-output operations

Parallel query processing can provide speed-up and scale-up. Speed-up is the reduction of response time, which you can accomplish by spreading the database over multiple partitions. Scale-up is the maintenance of a constant response time through database growth, which you can accomplish by increasing the number of partitions.

When tables and indexes are partitioned across multiple disks, SQL/MP can use a different process for each partition during query execution. This approach reduces the time for scans and other set-oriented processing by a factor equivalent to the number of partitions when there is no contention in the processor-controller disk path; that is, when every participating disk is primary to a different processor.

Parallel join operations are performed by the SQL executor during query processing.

Parallel index maintenance reduces the effect of multiple indexes on performance. Each index on a table is automatically updated whenever a row is inserted into the table and whenever a value is updated in any key column of the index. Multiple indexes on a table can be updated in parallel by different disk processes or by the same disk process servicing multiple requests.

Parallel index loading speeds the loading of a partitioned index by loading all partitions of the index at the same time. A different process loads each partition.

Parallel sorting is performed by the FastSort product, which SQL/MP uses for sort operations. You can configure subsort processes for parallel sort operations.

Parallel input-output operations are performed on multiple partitions by different disk processes. A single disk process can also perform parallel I/O by buffering operations in cache.

High Availability

SQL/MP has these features to help ensure high availability for databases:

- Online dumps using the TMFCOM DUMP FILES command, with complete support for TMF file recovery to recover a database in case of a disaster
- Online database reorganization capabilities such as online index creation with concurrent read and update capability; online partition moving; partition splitting;

and row redistribution with concurrent read and update capability available for the table or index

- Parallel table loads (using multiple SQLCI LOAD PARTONLY operations) and index loads (using CREATE INDEX or LOAD command with PARALLEL EXECUTION ON) to reduce the time required to load the object
- Automatic recompilation or partial recompilation, which eliminates the need to terminate program execution when changes in database structure or the environment make rebinding necessary
- Ability to defer name resolution in SQL statements until execution time

Database Management Operating Environments

SQL/MP is accessible to the Open System Services (OSS) and Guardian operating environments, as follows:

- SQL/MP data is accessible to programs that run in the OSS and Guardian environments.
- SQL tables, views, indexes, collations, catalogs, and SQL programs that are stored in Guardian files are manipulated by using SQLCI or Guardian utilities as described in this manual.
- SQL programs that are stored in OSS files are manipulated by using OSS utilities as described in the *Open System Services Shell and Utilities Reference Manual*.
- There are two SQL compiler interfaces, one for the OSS environment and one for the Guardian environment:
  - c89 is used in the OSS environment, supports OSS path names as input files, and produces OSS object files. For more information, see the *SQL/MP Programming Manual for C*.
  - SQLCOMP is used in the Guardian environment, supports Guardian file names as input files, and produces Guardian object files. For more information, see the SQL/MP programming manuals.
- The unique ZYQ name associated with an OSS path name can be used in some Guardian and SQLCI utilities. There is only one ZYQ name for each SQL program stored in the OSS environment, even if there are multiple links to the file.
- Guardian file names can be accessed from OSS—and supplied as input to c89—by using this format:

  ```
  /G/volume/subvolume/file
  ```
- An example of a file name follows:

  ```
  /G/data1/subvol1/testfl
  ```

- When referring to SQL objects from embedded SQL statements within an SQL program compiled for the OSS environment, use Guardian names or DEFINEs.

Database Management Tasks

To manage an SQL/MP database, you must perform all the tasks required to create the database, ensure its availability to users, and perform required changes. Because the database is an integral part of the application, measuring application performance and tuning the database configuration to enhance performance are also database management tasks.

SQL/MP database administrators (DBAs) perform these tasks:

- Install the SQL/MP software
- Upgrade to newer versions of SQL/MP software or downgrade to older versions
- Determine database layouts and data dictionary plans
- Plan for database security, integrity, and recovery
- Create and load databases
- Query catalog tables for information about databases
- Alter databases
- Manage databases and programs
- Reorganize and move databases
- Manage database applications
- Perform recovery operations
- Manage distributed databases
- Assemble and optimize queries
- Measure and enhance performance

Database Management Tools

SQL/MP provides the following types of SQL statements:

- Data Definition Language (DDL) statements that define and manage database objects.
- Data Manipulation Language (DML) statements that query and modify database tables.
- Data Control Language (DCL) statements and directives that control performance-related aspects of SQL/MP such as parallel processing and access paths.

- Data Status Language (DSL) statements that retrieve status information about catalogs and about versions of objects and software components.

In addition, SQL/MP supplies installation commands to install SQL/MP and a set of database management utilities.

DDL, DML, DCL, and DSL statements, Guardian utilities, and SQL utilities are the basic tools of the SQL/MP database administrator. All these statements and commands are available through the SQL/MP conversational interface (SQLCI).

Table 1-1 summarizes SQL/MP statements and commands for database management.

Note. For SQL programs in the OSS environment, OSS utilities are available that support DDL functions. SQLCI and the SQL utilities listed in Table 1-1 do not, in general, support SQL programs stored in OSS files, although exceptions are noted in descriptions of individual SQL utilities in this manual. For more information about OSS utilities, see the *Open System Services Shell and Utilities Reference Manual*.

Table 1-1. SQL/MP Statements and Commands for Database Management (page 1 of 3)

Statement or Command Type	Statement or Command	Description
DDL	ALTER	Alters the definition of a table; changes the attributes of an index or table; alters security attributes of a catalog, index, program, table, or view; renames an index, table, view, or SQL program stored in a Guardian file; adds, moves, or drops a partition of a table or index; splits a partition of a table or index.
	COMMENT	Adds a comment to an object definition.
	CREATE	Creates a catalog, constraint, index, table, or view.
	DROP	Drops a catalog, constraint, index, table, view, or SQL program stored in a Guardian file.
	HELP TEXT	Specifies help text for a column of a table or view.
	UPDATE STATISTICS	Updates information about the contents of a table and its indexes.
Operation on Other SQL Statements	DISPLAY STATISTICS	Displays information about the most recently executed DML or PREPARE statement.
Utilities	APPEND	Appends data to the end of a table or table partition, adding to existing data.
	CLEANUP	Deletes damaged SQL objects that result from corruption of definitions in the data dictionary.

Table 1-1. SQL/MP Statements and Commands for Database Management (page 2 of 3)

Statement or Command Type	Statement or Command	Description
	CONVERT	Converts an Enscribe file definition to an equivalent SQL CREATE TABLE statement and, optionally, generates SQLCI LOAD commands for loading data from the Enscribe file into an equivalent SQL table.
	COPY	Copies data to and from Enscribe files and SQL tables, adding to existing data.
	DISPLAY USE OF	Displays a list of SQL objects that depend on specified objects.
	DUP	Makes identical copies of Guardian files or of objects and object descriptions.
	EXPLAIN	Explains how data is accessed by specific DML statements and displays DEFINE mappings.
	FILEINFO	Displays physical characteristics of objects and files.
	FILENAMES	Displays a set of names that match a pattern specified with wild-card characters.
	FILES	Displays the names of files that are on one or more subvolumes.
	INVOKE	Generates a record description of a table or view.
	LOAD	Loads data into a file or table, overwriting existing data.
	MODIFY [DICTIONARY]	Modifies node numbers stored in file labels of SQL objects (LABEL option), modifies node names stored in SQL catalogs on the local node (CATALOG option), and registers user-defined catalogs in the local system catalog (REGISTER option).
	PURGE	Purges one or more objects or Guardian files.
	PURGEDATA	Deletes data from Guardian files and tables.
	SECURE	Changes the owner or security of one or more objects or Guardian files.
	UPGRADE CATALOG	Upgrades catalogs created with an older version of SQL/MP to the format required to use new features in a newer version.
	VERIFY	Checks the consistency and validity of object definitions in catalogs and file labels and displays names of invalid programs.

Table 1-1. SQL/MP Statements and Commands for Database Management (page 3 of 3)

Statement or Command Type	Statement or Command	Description
Installation	CREATE SYSTEM CATALOG	Creates the system catalog, including the CATALOGS table.
	DROP SYSTEM CATALOG	Deletes the system catalog, including the CATALOGS table, and deletes the SQLCI2 program.
	INITIALIZE SQL	Prepares a NonStop system to run SQL/MP.
DML	DELETE	Deletes rows from a table or view.
	INSERT	Inserts rows into a table or view.
	SELECT	Retrieves data from tables and views.
	UPDATE	Updates values in columns of a table or view.
DCL	CONTROL EXECUTOR	Specifies whether to process data by using a single executor or multiple executors working in parallel.
	CONTROL QUERY	Specifies optimizing queries either for the first few rows returned or for all rows returned.
	CONTROL TABLE	Specifies parameters that control locks and other attributes of tables.
	FREE RESOURCES	Releases locks on nonaudited and some audited objects and closes cursors.
	LOCK TABLE	Locks a table or underlying tables of a view and associated indexes.
	SHOW CONTROL	Displays the values in effect for any options of CONTROL statements.
	UNLOCK TABLE	Releases locks held on nonaudited tables or views.
DSL	GET CATALOG OF SYSTEM	Retrieves the name of a local or remote system catalog.
	GET VERSION	Retrieves the version of a specific SQL object, catalog, or software component.
	GET VERSION OF PROGRAM	Retrieves the PCV, PFV, or HOSV of an SQL program.

Through its Guardian environment, the NonStop OS provides additional utilities and subsystems for managing an SQL/MP database. Collectively, these utilities and subsystems are known as the NonStop Tools. You use these tools primarily in recovery operations, in moving a database, or in reorganizing a database online. Table 1-2 on page 1-14 summarizes these software products.

Table 1-2. NonStop Tools for Database Management

Program	Description
BACKUP	Copies Guardian files from disk to magnetic tape.
DataLoader/ MP	Loads and maintains SQL/MP databases (designed for large decision support system (DSS) tables).
DSAP	Disk Space Analysis Program: analyzes use of space on disk volumes, reporting on factors such as free space availability and extent allocation.
FILCHECK	Checks internal consistency of structured files and reports consistency errors.
FUP	File Utility Program: reorganizes key-sequenced files while they are in use, licenses programs, and obtains information about SQL objects and files.
Measure Product	Collects performance statistics on SQL objects, including information about processes, SQL statements, and file activity.
PERUSE	Invokes the PERUSE utility program.
PUP	Peripheral Utility Program: A utility used in D-series and earlier RVUs to manage disks and perform various operations on disk volumes in the SQL/MP database environment. In G-series RVUs, PUP functions are performed by the SCF.
RESTORE	Copies Guardian files from magnetic tape to disk. The files on tape must have been written by BACKUP.
Safeguard Product	Protects network and system resources, such as Guardian disk volumes, from unauthorized access.
SCF	Subsystem Control Facility: An interactive interface for configuring, controlling, and collecting information from a subsystem and its objects. SCF enables you to configure and reconfigure devices, processes, and some system variables while your HP NonStop S-series server is online.
TACL	Tandem Advanced Command Language, the command interpreter for the Guardian environment; provides an interactive user interface to a NonStop system through the operating system; enables users to run SQLCI and various system utilities, such as FUP, BACKUP, and RESTORE; and enables users to start SQL/MP processes.
TEDIT EDIT	Tandem PS Text Edit and EDIT text editor programs: used to create OBEY command files, generate SQLCI command strings, and read SQLCI log files.
VIEWSYS	Monitors disk input and output and cache memory use.

2 Installing SQL/MP

After ensuring that your node meets the hardware and software requirements for SQL/MP, you can install the SQL/MP relational database management system (RDBMS).

After installing SQL/MP, you can install the sample application distributed with the SQL/MP software, as explained in the *SQL/MP Reference Manual*. Use this application to demonstrate embedding SQL statements in host language programs, querying the catalogs, and querying and updating sample database tables. The sample database provided with this application is the same database used in examples in the SQL/MP manuals and education courses.

Note. If you are upgrading your SQL/MP environment from an older version of SQL/MP, consider the issues discussed in the *SQL/MP Version Management Guide* in addition to those discussed in this section before running your existing SQL/MP applications.

This section describes system requirements for SQL/MP, installation procedures, migration to a newer version, and the use of D-series features (for users migrating from a C-series node).

Hardware and Software Requirements

The hardware and software requirements for an SQL/MP RDBMS are:

- The hardware on which SQL/MP runs must be an SQL/MP system.
- Each system that includes SQL/MP requires a node (system) name, regardless of whether the node stands alone or is part of a network.
- The version of the operating system must be D30 or later to support versions 315 and newer of the SQL/MP software.
- As a general rule, TMF must be available when users are running SQL/MP application programs or using the conversational interface, SQLCI. The TMF subsystem is required for SQL compilation and for execution of all DDL statements, all SQL utilities, and DML statements that require TMF transactions for audited tables or views. TMF transactions are not required for previously compiled SQL statements that refer to nonaudited tables or views or for SELECT statements that use BROWSE ACCESS.
- The product version of the TMF subsystem must be D30 or later to support versions 315 and newer of SQL/MP software.
- All SQL objects must reside on volumes audited by the TMF subsystem. SQL programs need not reside on audited volumes.
- If you plan to use a national character (NCHAR) data type from SQL/MP, the system default multibyte character set must be a character set supported by SQL/MP. To check the default character set, use the Guardian MBCS_DEFAULTCHARSET_ procedure. To specify a different character set,

rename the appropriate LIBOBJ*nn* library object file to LIBOBJ before using SYSGEN to generate the node. SYSGEN accepts only one LIBOBJ file.

- For a list of supported character sets, see Defining Columns on page 5-19.

Recommendations for a node running SQL/MP are:

- Mirrored volumes are recommended, but not required, for volumes containing SQL objects.
- A minimum of 16 megabytes of memory is suggested for each processor in a node running SQL/MP. Additional memory can improve performance.

SQL/MP Software Components

The SQL/MP relational database management system consists of:

- BACKUP and RESTORE utilities
- Disk process
- FastSort software
- SQL catalog manager (SQLCAT)
- SQL compiler (SQLCOMP)
- SQL compiler interface
- SQL conversational interface (SQLCI)
- SQL executor
- SQL file system
- SQL utilities (SQLUTIL)
- Collation compiler
- Audit server (AUDSERV)

Installing SQL/MP

You should follow any instructions given in the associated software release document and any installation instructions that come with your site update tape (SUT). Install the software from the SUT by using the INSTALL program provided with the tape. The INSTALL program handles the installation of the SQL/MP system software on your $SYSTEM disk or alternate boot disk.

Starting the Transaction Management Facility (TMF)

Typically, you develop TMF startup and configuration files as OBEY command files. These files contain the TMF configuration options and parameters that describe the audit trails and the dump process. For information about TMF auditing requirements, configuration guidelines, and considerations for SQL/MP, see The TMF Subsystem on page 4-10 and the *TMF Planning and Configuration Guide*.

Your version of TMF must be compatible with your version of SQL/MP software as noted under Hardware and Software Requirements on page 2-1.

When you start the TMF subsystem, the configured data volumes are started for transaction processing if they are accessible. You must ensure the volumes you intend to use for SQL/MP catalogs and audited objects are configured and started for transaction processing. You must also ensure that the TMF subsystem as a whole is started for transaction processing. You can request status from TMFCOM, the TMF command interface, by entering the TMFCOM command at the operating system command interpreter prompt:

```
21> TMFCOM

~STATUS TMF                <--Request for TMF status.

TMF Status:
 System: \SQLNLS, Time: 4-Nov-1994 13:35:26
 State: started
 Transaction Rate: 5.0 TPS
AuditTrail Status:
 Master
  Active audit trail capacity used: 14%
  First pinned file: $DATA.ZTMFAT.AA000012
    Reason: Current file
  Current file: $DATA.ZTMFAT.AA000012
 BeginTrans Status: ENABLED
 Catalog Status:
  Status: up

~STATUS DATAVOLS           <--Request for status of data
volumes.

~status datavols
  Audit  Recovery
 Volume    Trail    Mode    State
 -------------------------------------------------------------
 $RAT      MAT    Online    Started
 $D00      MAT    Online    Started
 $XCEED    MAT    Online    Started
 $C30SYS   MAT    Online    Started
 $SYSTEM   MAT    Online    Started
 $SQL      MAT    Online    Started
 $TES      MAT    Online    Started

~EXIT                       <--Exits TMFCOM.
```

If you plan to use a system management program to operate TMF, you can use TMFSERVE, a TMF process that provides access to TMF by using the Subsystem Programmatic Interface (SPI). Both of these mechanisms can be used to monitor and control TMF operation.

For configuration information, see Guidelines for Configuring TMF on page 4-13.

Initializing SQL/MP

Before you can use SQL/MP for the first time, or when reinstalling the product, you must request initialization of SQL/MP by using the CREATE SYSTEM CATALOG and INITIALIZE SQL commands. The *SQL/MP Reference Manual* describes the syntax of these SQLCI commands.

SQL initialization involves creating the system catalog, SQL compiling the SQLCI2 program as a valid SQL program, then registering the program in the system catalog. The SQLCI2 program, when in execution, serves as a backend process for SQLCI.

The system catalog is like any other catalog, except it contains an additional table, CATALOGS, which is the system directory of catalogs. The CATALOGS table must reside on a subvolume named SQL.

By default, the system catalog resides on the subvolume $SYSTEM.SQL; however, you can specify another volume and subvolume in the CREATE SYSTEM CATALOG command. If the system catalog is not on the subvolume SQL, the CATALOGS table is placed on the same volume as the system catalog, but on a subvolume named SQL.

For information about locating and securing catalogs, see Creating Catalogs on page 5-1.

Initialization Steps

To initialize SQL/MP, follow these steps:

1. Check that the TMF subsystem is configured correctly and started.
2. Check that the $SYSTEM.SYSTEM.ZZSQLCI2 file exists on the node and save a copy. This temporary file, which contains the SQLCI2 program, is copied onto the disk by the INSTALL program.

   Note. As a basic rule, HP recommends you save a copy of the $SYSTEM.SYSTEM.ZZSQLCI2 (SQLCI2) program at all times. After the SQL initialization is complete (Step 6), the temporary copy (ZZSQLCI2) is renamed and becomes the permanent SQLCI2 program. If a subsequent SQL initialization is attempted or if the SQLCI2 program is corrupted or purged, the saved copy provides a backup.

3. Log on as the local super ID. The super ID is required to run the CREATE SYSTEM CATALOG and INITIALIZE SQL commands in Step 5 and Step 6 on page 2-6.
4. Start the SQLCI program. At the command interpreter prompt, enter the program name:

   ```
   23> SQLCI
   ```

5. If you are installing SQL for the first time on this node, create the system catalog. If you are reinstalling it, you can skip this step.

By default, the system catalog resides on the subvolume $SYSTEM.SQL; however, you can specify an alternative volume and subvolume.

The system catalog contains a table called CATALOGS, which is the system directory of catalogs. If you put the system catalog on a volume other than $SYSTEM, SQL/MP puts the CATALOGS table on a subvolume named SQL on the same volume as the rest of the system catalog.

Note. It is recommended that you do not place the system catalog on the $SYSTEM volume. When the system catalog resides on another volume, the $SYSTEM volume can function as a nonaudited volume and can also be rebuilt from a system image tape (SIT), in case of disaster, without affecting the SQL/MP catalog structure.

If you want the system catalog to reside on the default location ($SYSTEM.SQL), enter this command at the SQLCI prompt:

```
>> CREATE SYSTEM CATALOG;
```

If you want the system catalog to reside on a volume or subvolume other than the default location, enter this command at the SQLCI prompt:

```
>> CREATE SYSTEM CATALOG $vol.subvol;
```

In this command, $*vol.subvol* is a Guardian volume and subvolume name. If you do not specify a subvolume, the RDBMS uses the name SQL by default. If you specify a subvolume other than SQL, the RDBMS places all system catalog tables except CATALOGS on the subvolume you specify and places the CATALOGS table on a subvolume named SQL, on the same volume as the other catalog tables.

If you are installing SQL/MP on a system using the HP NonStop Storage Management Foundation (SMF), and you want to ensure that you can fall back to a non-SMF system, you should make sure that the system catalog tables reside on one physical volume. If you specify a virtual volume for the system catalog, SMF can distribute the system catalog tables among multiple physical volumes in the storage pool. When this configuration is in place, there is no guarantee that you can return to using a nonvirtual volume. When you are certain you will not need to fall back to a non-SMF system, you can specify a virtual volume for the system catalog tables without being concerned with the physical location of the files.

To ensure that the system catalog tables reside on one physical volume, you can specify a direct volume that is not in any storage pool, or you can use the PHYSVOL option, as follows:

```
>> CREATE SYSTEM CATALOG $virtual_vol.subvol
       PHYSVOL $physical_vol;
```

With the PHYSVOL option, you can specify only the volume name. Also, the virtual volume specified with the CREATE SYSTEM CATALOG clause must be associated with the same storage pool that contains the physical volume specified with PHYSVOL. For more information about using this option, see the *SQL/MP Reference Manual* and the *Storage Management Foundation User’s Guide*.

6. Initialize SQL/MP by entering this command at the SQLCI prompt:

   ```
   >> INITIALIZE SQL;
   ```

   During the installation of the SQL/MP system software, the INSTALL program places the new SQLCI2 program on the $SYSTEM.SYSTEM subvolume in the file named ZZSQLCI2. SQLCI2 is the process through which the SQL/MP conversational interface (SQLCI) communicates with the SQL/MP executor to request various functions.

   The SQL initialization process requested in this step drops the older version of SQLCI2 if it exists on the system, renames the ZZSQLCI2 file to SQLCI2, SQL compiles the program in $SYSTEM.SYSTEM.SQLCI2, and registers the program in the PROGRAMS table of the system catalog. To run SQL statements from SQLCI, SQLCI2 must be a valid, registered SQL program.

   The INITIALIZE command performs the installation operations automatically. You can request these operations directly, however, by entering commands at the command interpreter prompt (for installation on $SYSTEM.SYSTEM):

   ```
   24> PURGE $SYSTEM.SYSTEM.SQLCI2    (if you are reinstalling SQL)
   25> RENAME $SYSTEM.SYSTEM.ZZSQLCI2, $SYSTEM.SYSTEM.SQLCI2
   26> SQLCOMP/IN SQLCI2/ CATALOG system-catalog
   ```

   In the SQLCOMP command, *`system-catalog`* is the Guardian name of the system catalog.

7. Terminate SQLCI and create a backup copy of the collation compiler, which is in the $SYSTEM.SYSTEM.NLCPCOMP file:

   ```
   >> EXIT;
   26> FUP DUP $SYSTEM.SYSTEM.NLCPCOMP, &
   26> & $VOLBK.SUBVBK.NLCPCOBK, SOURCEDATE
   ```

   (The collation compiler translates character processing rules specified in a source file into an internal format.)

8. Use the SQLCOMP command to SQL compile the collation compiler:

   ```
   27> SQLCOMP /IN $SYSTEM.SYSTEM.NLCPCOMP,NOWAIT/ CATALOG
   $VOL1.SQL
   ```

   In this command, $VOL1.SQL is the subvolume where the system catalog resides

   Note. You must SQL compile the collation compiler manually anytime you install a software product revision (SPR) to the T6570 - National Language Character Processing product.

9. If you are installing SQL for the first time on this node, set the appropriate security string for the system catalog. You can use the ALTER CATALOG statement to alter the owner ID and security string of all the system catalog tables at once. Then, you

can use ALTER TABLE statements to resecure the CATALOGS, USAGES, TRANSIDS, and PROGRAMS tables separately.

```
>> ALTER CATALOG system-catalog SECURE "NG--";
```

```
>> ALTER TABLE system-catalog.CATALOGS SECURE "NA--";
```

For more information, see Securing the System Catalog on page 5-10.

10. Verify that these programs are secured for execute access on your node:

```
$SYSTEM.SYSTEM.SQLCI
$SYSTEM.SYSTEM.SQLCI2
$SYSTEM.SYSTEM.SQLCAT
$SYSTEM.SYSTEM.AUDSERV
$SYSTEM.SYSTEM.SQLCOMP
$SYSTEM.SYSTEM.SQLESP
$SYSTEM.SYSTEM.SQLESPMG
$SYSTEM.SYSTEM.SQLUTIL
$SYSTEM.SYSTEM.NLCPCOMP
$SYSTEM.SYSnn.RECGEN
$SYSTEM.SYSnn.SORTPROG
```

Note. SQLCI2 and NLCPCOMP must have both read and execute access.

In the preceding list, *nn* represents two digits assigned during node generation (SYSGEN operation).

Verify that this file is secured for read access on your node:

```
$SYSTEM.SYSTEM.SQLMSG
```

If your node is part of a network, check that the programs in the preceding list are secured for network execute access and that the $SYSTEM.SYSTEM.SQLMSG file is secured for network read access.

11. Verify that these Guardian utilities and SQL/MP components are licensed so that they can access SQL/MP objects. The INSTALL process, when completed normally, performs the licensing of these programs (automatically or manually):

```
$SYSTEM.SYSTEM.SQLCOMP
$SYSTEM.SYSTEM.SQLUTIL
$SYSTEM.SYSTEM.SQLCAT
$SYSTEM.SYSTEM.AUDSERV
$SYSTEM.SYSnn.BACKUP
$SYSTEM.SYSnn.DSAP
$SYSTEM.SYSnn.FUP
$SYSTEM.SYSnn.FILCHECK
$SYSTEM.SYSnn.PUP
$SYSTEM.SYSnn.SCF
$SYSTEM.SYSnn.RESTORE
$SYSTEM.SYSnn.SORTPROG
```

In the preceding list, *nn* represents two digits assigned during system generation (SYSGEN operation).

You can use the FUP SECURE command to alter the security of these programs and the FUP LICENSE command to license the programs, if necessary. To alter the security of the SQL sensitive program SQLCI2, you can either use the SQL ALTER PROGRAM statement or the FUP SECURE command.

Note. The SQLCI2 program is not licensed under normal circumstances. Only the super ID can license the SQLCI2 program. For more information, see Appendix A, Licensed SQLCI2 Process.

---

1. If old versions of SORT, SORTPROG, and RECGEN programs still exist on $SYSTEM.SYSTEM, remove them now by entering this command at the command interpreter prompt:

   ```
   26> PURGE $SYSTEM.SYSTEM.SORT, $SYSTEM.SYSTEM.SORTPROG,
             $SYSTEM.SYSTEM.RECGEN
   ```

   If SORT, SORTPROG, or RECGEN programs exist on $SYSTEM.SYSTEM from a previous RVU, the results are unpredictable.

   SQL/MP uses the FastSort programs SORT and SORTPROG for sorting operations and the RECGEN program for parallel loading of indexes. The INSTALL process moves these programs to the SYS*nn* subvolume (in which *nn* is two digits assigned during system generation).

2. Verify that the SORT, SORTPROG, and RECGEN programs on the SYS*nn* subvolume are secured for execute access on your node. If your node runs in a network, check that these programs are secured for network execute access. If SQL/MP initiates a SORT operation and the SORT, SORTPROG, or RECGEN security does not allow access, a run-time sort error, Sort Start Error 4, is generated.

   You can use the FUP SECURE command to alter the security of these programs, if necessary.

3. Check that the SQLMSG file you are using is the one released with the new version of SQL/MP. The message file contains error messages, warning messages, and help text for SQL/MP. You can check the version by running SQLCI and entering the ENV command; the message file version is listed as MESSAGEFILE VRSN in the ENV output.

   Check that these message files are secured for read access on your node:

   ```
   $SYSTEM.SYSTEM.SQLMSG
   $SYSTEM.SYSTEM.NLCPMSG
   ```

   If your node is part of a network, check that these message files are secured for network read access. If necessary, secure them for network access by entering these commands:

   ```
   32> FUP SECURE ($SYSTEM.SYSTEM.SQLMSG, &
   32> & $SYSTEM.SYSTEM.NLCPMSG), “NN-N”
   ```

4. If you use COBOL with SQL/MP, replace the COBOLEXT file on your $SYSTEM.SYSTEM subvolume with the correct COBOL extension file. The $SYSTEM.SYSTEM.COBOLEXT extension file installed by the INSTALL program does not contain the complete COBOL extension libraries for SQL/MP, which are stored under the name $SYSTEM.SYSTEM.COBOLEX0.

   To replace the existing $SYSTEM.SYSTEM.COBOLEXT file with the appropriate COBOL extension file, enter this command:

   ```
   27> FUP DUP $SYSTEM.SYSTEM.COBOLEX0, $SYSTEM.SYSTEM.COBOLEXT,
   PURGE, SOURCEDATE
   ```

   If an incorrect extension file is used for programs containing SQL statements, those programs might encounter a compilation error (UNIT OF PROPER LANGUAGE NOT FOUND) or a run-time trap error.

Setting Up Event Logging

SQLCI provides logging capability to:

- Log certain events to a terminal or file automatically
- Log command strings entered interactively through SQLCI, and, optionally, output from those commands, to an EDIT file

Both methods are effective for maintaining a record of events and commands. You should routinely have the log file duplicated, printed, and cleared. For more information about event logging, see the discussion of the =_SQL_CMP_EVENT DEFINE in the *SQL/MP Reference Manual*.

Setting Up Alternate SQL Components

Users with the super group ID can specify alternate SQL components through the =_SQL_*`component`* DEFINEs. These DEFINEs redirect the RDBMS to use the specified programs or message files instead of the default programs or files residing on $SYSTEM.SYSTEM. The use of these components is limited to members of the super group only.

If you want an SQL/MP system available only for the super group, issue the =_SQL_*`component`* DEFINEs before performing the initialization or reinstallation steps. You might do this, for instance, when you want to allow a limited group of users to access a customized SQL/MP system or to run SQL components not residing on $SYSTEM.SYSTEM. For more information on the =_SQL_*`component`* DEFINEs, see the *SQL/MP Reference Manual*.

Additional Installation Considerations

Installation of SQL/MP might require installation of a specific version of related software. For versioning requirements of products associated with SQL/MP, check the documentation supplied with your software.

If you install an SPR to the T6570 - National Language Character Processing product, you must SQL compile the collation compiler manually. For more information, see Step 8 on page 2-4.

If you plan to define large numbers of partitions for tables and indexes in the database, consider using the =_SQL_CAT_HEAP_LIMIT DEFINE to increase the heap space size limit for the catalog manager process. For more information, see the *SQL/MP Reference Manual* or SQLCI online help. You can specify this DEFINE in the TACLCSTM file for TACL sessions running applications or interactive queries.

Reinstalling SQL/MP Software

To reinstall SQL/MP on a node that has previously run the software, follow the steps listed under Installing SQL/MP on page 2-2.

Note that if you have installed SQL/MP on your node before, you do not need to re-create the system catalog.

Migrating to a Newer Software Version

To replace an existing version of the SQL/MP software, install the newer version of the software as described under Installing SQL/MP on page 2-2.

After you install the new version of SQL/MP software, follow these guidelines:

- Continue to run existing applications without recompilation.
- Continue to use existing local user catalogs until you are ready to start using the new features provided with the newer version of SQL/MP. You can continue to use your existing system catalog indefinitely, unless you plan to register newer version objects in the system catalog.
- Continue to access all local and remote objects in your database, except for remote objects registered in catalogs whose versions are newer than the version of the SQL/MP software on either the local or remote node. For information about managing a network that has multiple versions of software, catalogs, and objects, see the *SQL/MP Version Management Guide*.
- Test the newly installed software before you use new features. This evaluation might require running the new software for several days of testing or simply running your applications under the new software for however long is appropriate. You can also test the new software by using the sample database distributed with your older RVU or by using your own test database.

- Test the features available with the newer version of the software. Create test catalogs and objects for testing purposes, rather than altering existing objects. For example, collations require a version 300 or newer catalog to register the collation. Before adding a collation, create a test catalog to associate with the collation. Similarly, do not add a column with a new data type to an existing production table, because the table will not be accessible if you need to revert to an older software version.

  If you need to revert to the older RVU, you can do it easily at this point because the versions of production catalogs and objects are unchanged, and there are no incompatible TMF audit records. (Guidelines for reverting are discussed under Reverting to an Older Software Version on page 2-15.)

- When you migrate to an RVU that supports SQL Format 2-enabled tables and the SQL Format 2 partitions they contain, the only consideration that might affect you is which RVU you migrate from. For more information on planning for Format 2 partitions, operational and fallback considerations, and interoperability, see Appendix C, Format 2 Partitions.

Do not perform any of these actions until you are reasonably sure you do not want to return to the older version of SQL/MP software:

- Create a production catalog whose version is newer than your previous version of the SQL/MP software. A newer version catalog is created automatically when you create a catalog on a node running the new version of the software.
- Create a production table using features that cause the version of the table to be newer than your previous version of SQL/MP software.
- Compile a program using the newer version of the SQL/MP compiler. If you do recompile a program and decide to return to an older version of the software, you must recompile the program with the older version of the SQL compiler after reverting back to the older version. If you decide to compile programs with the newer version of the software, keep track of which programs you compile.

When you are reasonably sure you do not want to revert to the older version of the software, perform these steps as needed, based on your use of new features available in the newer version of the software:

1. Upgrade catalogs. After you finish testing the newer version of the SQL/MP software, upgrade the user catalogs on your node as needed to use version 300 or newer features.

   Note. You do not need to upgrade user catalogs until you are ready to register objects or programs with newer-version features. You can also continue to use your older-version system catalog indefinitely unless you register newer-version user objects or programs in the system catalog, because newer-version user catalogs can be registered in a older-version system catalog. For example, you can register version 300 catalogs in a version 1 or version 2 system catalog. You cannot, however, register version 300 or newer objects in a version 1 or version 2 system catalog.

If you are migrating from version 1 to a version 300 or newer version, you can use these version 2 features without upgrading any version 1 catalogs: parallel execution, local autonomy, parallel index maintenance, and virtual sequential block buffering.

You can have a combination of versions of catalogs on a node. Unless you have an important reason for keeping this combination (such as operating in a mixed-version network), you should eventually upgrade all the catalogs on the node to simplify your database management efforts.

For additional information, see Upgrading Catalogs on page 2-14.

2. Recompile applications if desired. You do not need to SQL compile your application programs unless you want to take advantage of new features or performance enhancements.

   To take advantage of new features, you might have to modify your programs. For example, versions 325 and newer software support CASE expressions. To use them, you need to modify source code and then host-compile and SQL compile the modified programs by using the newer versions of the SQL/MP compilers. For more information, see the *SQL/MP Programming Manual*.

   To take advantage of performance enhancements, host-compile and SQL compile existing programs to ensure that the execution plan is optimal for the new SQL/MP software. For example, versions 310 and newer SQL compilers generate query execution plans that provide better performance than plans generated by older-version SQL compilers.

   With the exception of a dynamic SQL program that does not use the RELEASE option, a program written for an older version of the SQL/MP software should compile and execute on a node that has version 310 or newer software with no source code changes.

C-Series to D-Series Migration Considerations

If your previous installation of SQL/MP was on a node that ran a C-series RVU of the operating system, you should be aware of differences between the C series and D-series systems that affect the operation of SQL/MP.

The D-series features that might affect your SQL/MP applications are high and low process identification numbers (PINs), changes to file naming rules, and subvolume defaulting. In addition, you must understand the rules for combining C-series and D-series object modules.

For more information about D-series features, see the *D-Series System Migration Planning Guide*.

Mixed-Version Network Considerations

SQL/MP processes run at a high PIN by default. However, a process that must access objects on C-series nodes must not run at a high PIN. Therefore, these SQL/MP processes must run at low PINs if they communicate with a process on a C-series node (or any process that cannot respond to a high-PIN process):

- FastSort (RECGEN, SORT, SORTGEN)
- SQL catalog manager (SQLCAT)
- SQL compiler (SQLCOMP)
- SQL conversational interface (SQLCI)
- SQL executor server processes (SQLESP)
- SQL utilities (SQLUTIL)
- SQLCI2
- Collation compiler (NLCPCOMP)

There are several approaches for running SQL programs at low PINs, including these:

- Setting the TACL HIGHPIN environment variable to OFF when running any SQL program:
    - Arrange to have all TACL processes inherit a HIGHPIN OFF setting, either by specifying SET HIGHPIN OFF in the initial TACL process (before starting all other TACL processes on the node) or by placing a SET HIGHPIN OFF command into the $SYSTEM.SYSTEM TACLCSTM file, executed by every TACL process during logon.
    - Alternatively, specify HIGHPIN OFF as needed in TACL RUN commands or SQLCI OBEY command files.

  Remove the HIGHPIN commands when access to C-series nodes is no longer needed.

- Setting the HIGHPIN attribute to OFF in all object files by using the BIND CHANGE command to change the HIGHPIN attribute for SQL program files after they have been installed. The syntax is:

  ```
  BIND CHANGE HIGHPIN OFF IN $SYSTEM.SYSTEM.<filename>
  ```

  where <filename> is SQLCI, SQLCI2, SQLCAT, SQLUTIL, SQLCOMP, SQLESP, RECGEN, and NLCPCOMP.

  Before making this change, note the license flag setting for each file and relicense any licensed files after changing the HIGHPIN attribute. If you use this approach, continue to change the HIGHPIN setting after any subsequent installation of SQL software until access to C-series nodes is no longer needed.

For parallel index creation or parallel index load operations that access base table partitions on nodes running older versions of SQL/MP, specify the LOCALONLY option at the start of the CREATE INDEX configuration file to force RECGEN processes to run on the local node. Alternatively, start the CREATE INDEX operation from the node running the older version of software.

△ **Caution.** If an SQL object has the UNRECLAIMED FREESPACE (F) or INCOMPLETE SQLDDL OPERATION (D) attribute set, do not attempt to back up, move, or duplicate the object until the attribute is reset. For more information, see UNRECLAIMED FREESPACE (F) and INCOMPLETE SQLDDL OPERATION (D) Flags on page 7-24.

Upgrading Catalogs

You must upgrade or create at least one user catalog on a node running the new version of SQL/MP software before you can do any of these:

- Create an object with attributes that cause the object version to be newer than the current version of the user catalog. For example, you cannot create an object that refers to a version 320 feature and registers the object in a version 1 or version 2 catalog.
- Alter an object by adding attributes that cause the object version to be newer than the current version of the user catalog. For example, you cannot add a column with a version 310 data type to a table registered in a version 1 or version 2 catalog.
- Register programs with a program catalog version newer than the current version of the user catalog. For example, you cannot register a program that contains a version 320 feature in a version 1 or version 2 catalog.

To create a version 300 or newer catalog, use the CREATE CATALOG command.

To upgrade an existing user catalog to version 300 or newer, use the UPGRADE CATALOG command. This command requires exclusive access to the catalogs being upgraded and to the objects to be registered in the catalogs. Specifically, to use UPGRADE CATALOG, you must be the local owner of the catalog, the local super ID, the group manager, or the remote owner with authority to purge the catalog tables. You must also have authority to write to the CATALOGS system catalog table.

SQL/MP automatically requests the TMF subsystem to protect the integrity of the database during the upgrade operation. If a user-defined TMF transaction (a transaction explicitly defined by using language statements such as BEGIN WORK and COMMIT WORK) is not in progress when you enter the UPGRADE CATALOG command, SQL/MP begins and ends a TMF transaction for each catalog being upgraded.

For statements issued within a user-defined TMF transaction, SQLCI does not initiate a system-defined TMF transaction. You should allow SQLCI to initiate TMF transactions for DDL commands.

You can use a single UPGRADE SYSTEM CATALOG command to upgrade all the catalogs on a node. However, doing so can take several minutes, and during this time, no user or program can access any catalogs or objects on the node. For this reason, you might want to upgrade catalogs individually.

This example upgrades three user catalogs, one catalog at a time:

```
>> UPGRADE CATALOG \SYS1.$VOL1.SALES TO 310;
--- SQL operation complete.
>> UPGRADE CATALOG \SYS1.$VOL1.INVENT TO 310;
--- SQL operation complete.
>> UPGRADE CATALOG \SYS1.$VOL1.PERSNL TO 310;
--- SQL operation complete.
```

You can continue using the existing system catalog and create versions of catalogs that are newer than the current system catalog (but not newer than the SQL/MP software version on the node on which the catalog resides). For example, you can create a version 310 catalog to be registered in a version 1 or version 2 system catalog residing on a version 310 SQL/MP node.

If you decide to register objects and programs in the system catalog, instead of in user catalogs, you must upgrade the system catalog before registering objects or SQL programs that have a newer version than the system catalog. To do this, use the UPGRADE SYSTEM CATALOG command, identifying the system catalog as the one to be upgraded.

△ **Caution.** Because of database administration overhead, you should not register user objects in the system catalog. Also, the impact of upgrading a system catalog in a mixed-node environment can make this system inaccessible to other nodes in the network.

For the syntax and other examples of UPGRADE CATALOG and UPGRADE SYSTEM CATALOG, see the *SQL/MP Reference Manual*. For more information about catalog versions, see the *SQL/MP Version Management Guide*.

Reverting to an Older Software Version

If you have upgraded the system catalog to a newer version and you want to revert to an older version of SQL/MP, you must first drop all objects whose versions are newer than the version of SQL/MP to be reinstalled. You must then downgrade the system catalog and any user catalog with a version newer than the version of SQL/MP to which you are reverting. After you reinstall the older version of SQL/MP software, you must recompile newer-version programs.

Dropping Newer-Version Objects

To preserve data in tables that have a newer version than the software to which you are reverting, you must create each of the tables again, omitting any features that caused the table to have the newer version. For example, if tables use extended partition arrays, re-create them with standard partition arrays. If the tables you want to preserve are defined with new features, write a program to move the required data from the newer-version tables to the older-version tables or use SQLCI for this purpose. You must perform these tasks before installing the older-version software.

△ **Caution.** If any tables have the UNRECLAIMED FREESPACE (F) or INCOMPLETE SQLDDL OPERATION (D) attribute set, remedy the situation before downgrading the version. Otherwise, the table might be corrupt after the downgrade is completed. For more information, see UNRECLAIMED FREESPACE (F) and INCOMPLETE SQLDDL OPERATION (D) Flags on page 7-24.

After preserving any necessary data, you can drop all newer-version objects. You can also drop any programs that must be registered in a newer-version catalog; this includes any program with a newer-version feature, which causes the program catalog version to adopt the newer version.

Downgrading Catalogs

If you need to access objects in a newer-version user catalog, you must downgrade the user catalog before installing the older version of SQL/MP software.

You must always downgrade a version 300 or newer system catalog when you revert to an older version of the software, but you do not always need to downgrade version 300 user catalogs. For example, if you are temporarily reverting to version 2 from version 310 and do not intend to access objects registered in user catalogs, you can leave any version 310 user catalogs and objects on the node and downgrade only the system catalog.

Note. You do not need to downgrade any version 2 user or system catalogs, because version 2 catalogs are compatible with version 1 SQL/MP software.

If, however, you need to access data from the version 310 user catalogs on the node where you are reinstalling an older version of SQL/MP, you must downgrade these catalogs. If you do not downgrade the catalogs on a node in which the SQL/MP software reverts to an older version, a fallback situation results. In this situation, the reinstalled SQL/MP software cannot access any catalogs whose versions are newer than the version of the software or any objects or programs registered in the catalogs. In addition, SQL/MP software on other nodes in a network can no longer access these catalogs and objects, regardless of the versions of the software on the remote nodes.

△ **Caution.** If you want to downgrade any catalogs, you must do so before you reinstall the older version of the SQL/MP software, because only version 300 or newer software supports the DOWNGRADE CATALOG and DOWNGRADE SYSTEM CATALOG commands.

Downgrading User Catalogs

Before downgrading a user catalog, you must:

- Drop any protection views on the catalog tables to be downgraded.
- Drop any object whose version would be newer than the version of the catalog after the catalog is downgraded.

- Drop any program whose program catalog version would be newer than the version of the catalog after the catalog is downgraded. For example, drop version 310 and newer programs before reverting to version 2.

You cannot downgrade a catalog to version 1, but version 2 catalogs are compatible with version 1 SQL/MP software if you access only version 1 objects.

For information about catalog, object, and program versions, see the *SQL/MP Version Management Guide*.

You must have exclusive access to the catalog tables. You can downgrade more than one catalog by specifying a catalog name pattern. For a detailed description of the DOWNGRADE CATALOG command and the requirements for using it, see the *SQL/MP Reference Manual*.

Downgrading the System Catalog

Before you can downgrade the system catalog to a version older than 300, you must drop any version 300 or newer objects registered in the system catalog. You must be logged on with the super ID.

You cannot downgrade the system catalog to version 1, but version 2 system catalogs are compatible with version 1 SQL/MP software.

For a description of the syntax and more examples of using the DOWNGRADE SYSTEM CATALOG command, see the *SQL/MP Reference Manual*.

Recompiling Programs

If you want to use a program compiled with a version 300 or newer SQL compiler (a program whose program format version is 300 or newer), you must recompile the program with the older-version SQL host compiler compatible with the version of SQL/MP you have reinstalled, and then SQL compile the program with the appropriate SQL compiler.

Reverting to SQL/MP Version 2

To revert to version 2 of SQL/MP:

1. If you have any version 300 or newer tables you want to preserve, use the CREATE TABLE statement to create version 2 tables to contain any version 2 data you want to preserve, and use the LOAD utility to move the data from the version 300 tables to the version 2 tables.
2. Use the DROP statement to drop version 300 or newer tables, indexes, views, and collations.

3. Make a list of all programs whose program format version or program catalog version is 300 or newer. You can generate a list by entering a SELECT statement like the following for each catalog on the node:

   ```
   >> SELECT PROGRAMNAME, PROGRAMFORMATVERSION,
   +>   PROGRAMCATALOGVERSION
   +> FROM $CATVOL.SYS.PROGRAMS     <--PROGRAMS table in system catalog
   +> WHERE PROGRAMCATALOGVERSION >= 300
   +> OR PROGRAMFORMATVERSION >= 300 ;
   --- SQL operation complete.
   ```

   To inquire about the version of a specific program, use the GET VERSION OF PROGRAM statement.

4. Use the DROP PROGRAM statement (or the corresponding OSS utility for OSS programs) to drop all programs whose program catalog version is 300 or newer.

5. Use the DOWNGRADE CATALOG command to downgrade all version 300 or newer user catalogs to version 2:

   ```
   >> DOWNGRADE CATALOG catalog-name TO 2 ;
   --- SQL operation complete.
   ```

6. If the system catalog is version 300 or newer, use the DOWNGRADE SYSTEM CATALOG command to downgrade the system catalog to version 2:

   ```
   >> DOWNGRADE SYSTEM CATALOG TO 2 ;
   --- SQL operation complete.
   ```

7. Reinstall version 2 of the SQL/MP software as described under Reinstalling SQL/MP Software on page 2-10.

8. If any programs you want to use were host-compiled by a version 300 or newer host compiler, use a version 2 host compiler to recompile the source code for these programs.

9. If any programs you want to use have a program format version of 300 or newer (as determined in Step 3), use a version 2 SQL compiler to recompile these programs.

Reverting to an Older Version of TMF

When moving from version 300 or newer SQL/MP software back to an older version of software, you must also revert back to the older version of TMF software. For more information, see the *TMF Planning and Configuration Guide*.

3 Understanding and Planning Database Tables

An understanding of the types of organizations of tables and their corresponding file structures is essential for effective use and functioning of the database.

This section presents conceptual information about tables and how to plan table organization. Subsequent sections describe how to create tables and associated database objects.

Understanding SQL File Structures

A relational database consists of two structural levels, the logical and the physical:

- The logical level includes the tables and views you access directly through SQL statements. When you request an operation on the database or a display of its contents, you work with the database at the logical level.
- The physical level underlies the logical level and is composed of physical files on disks.

SQL tables and indexes are implemented on the physical level as Guardian files. If a table or index is partitioned, it is implemented as multiple separate files. Table and index files are managed by the SQL file system and are accessed implicitly through the DP2 disk process.

The data you insert into tables and views is stored in these underlying files. Tables and indexes are associated with their corresponding physical files through entries in the data dictionary for your database. Views are associated with physical files through their underlying tables.

When you create a table, you establish the characteristics of the underlying file by specifying them as parameters in the CREATE TABLE statement. That is, you create the file implicitly through this statement rather than directly through an explicit file-creation statement.

The type of table created determines the type of the corresponding file:

- A key-sequenced table has a key-sequenced file structure.
- An entry-sequenced table has an entry-sequenced file structure.
- A relative file has a relative file structure.

Data-transfers to and from the files are done in terms of logical records and key fields within those records.

Each file has a unique primary key associated with it, which contains a unique value used to order and identify records in the file.

The following text describes primary keys and then discusses the structures of the three types of files. An understanding of these structures can help you plan for the best use of disk storage space when sizing your database, implementing economical table access methods, and analyzing various performance trade-offs. This understanding is also essential for anyone using the FILCHECK or TANDUMP utility to operate on physical file structures.

Primary Keys

All SQL/MP table organizations have a unique primary key. Key values affect the order in which rows are stored and retrieved. The primary key also serves as the physical primary key for the DP2 disk process.

Note. In this discussion of keys, do not confuse the physical primary key with the logical SQL primary key. The differences are:

- The physical primary key is the key used by the DP2 disk process to access records within the file. Within a record, this key must be unique. A key-sequenced file has only one physical primary key.
- The primary key is the key recognized and used at the logical level by ANSI standard SQL. This key is specified in the PRIMARY KEY clause of the CREATE TABLE statement or partially specified in the CLUSTERING KEY clause. The value of the primary key must be unique for each row. If you specify a PRIMARY KEY clause, the physical primary key is also based upon this specification. But if you do not specify a PRIMARY KEY clause, the RDBMS generates part or all the physical primary key.

At the physical level, a key is a field or group of fields the system can use to order records and to identify records for processing. The primary key can be user-defined, system-defined, or a combination of both.

User-Defined Primary Key

A user-defined primary key consists of one or more columns (fields) whose values uniquely identify the rows of a table and determine the order in which the rows are stored. The combined value of all primary-key columns must be unique for each row in the table. You can define the primary key by using the PRIMARY KEY clause in the CREATE TABLE statement.

Each column has an ordering attribute, either ASCENDING or DESCENDING, that determines the order for storing and retrieving the rows.

For multicolumn (composite) keys, if multiple rows share the same value for the first key column, the value in the second key column is used to determine the order for storing or retrieving the rows. If multiple rows share the same values for the first and second key columns, the third key column is used to determine the order, and so forth.

The length of a primary key cannot exceed 255 bytes. To calculate the length, find the sum of the lengths in bytes of the columns that make up the key. For the length of a varying-length column (VARCHAR, NCHAR VARYING), use the number of bytes defined for the column only; do not include the two-byte descriptor. In a primary key for a table, a column cannot contain the null value; the column is implicitly defined as NOT NULL.

User-defined primary keys facilitate generic locking as explained under Using Generic Locks on page 14-21.

System-Defined Primary Key

A system-defined primary key consists of a column named SYSKEY, generated and maintained internally by SQL. For each row (record), the SYSKEY column contains a unique system-generated value. The SYSKEY value alone serves as the primary key for a key-sequenced table when no primary key is specified by the PRIMARY KEY clause and no clustering key is specified by the CLUSTERING KEY clause.

When a table is stored on the basis of SYSKEY values, the rows are not stored on the disk in an order that is particularly useful, except for queries that scan rows in the order of their entry. Consequently, queries often result in scans of the entire table. Alternatively, when an index is used, queries that access a range of rows are not processed efficiently because they require multiple random accesses to the table. To improve efficiency in the absence of a primary key, you can physically order and access the table by using a clustering key (described in Clustering Key Combined With System-Defined Key).

The SYSKEY column has the data type LARGEINT SIGNED. The value is a unique eight-byte number generated by the system on the basis of timestamps. SYSKEY is physically created as the first column of the table. The description of the table in the catalog reflects the presence of the SYSKEY column.

Inserts into the file cannot specify a value for SYSKEY, and SYSKEY values cannot be updated.

A key-sequenced table with a system-defined primary key cannot have partitions.

Clustering Key Combined With System-Defined Key

A combination key consists of a user-defined clustering key, with columns specified in the CLUSTERING KEY clause of the CREATE TABLE statement and a system-defined SYSKEY column. When you use the CLUSTERING KEY clause, SQL/MP appends the ascending system-defined SYSKEY column to the last column of the potentially nonunique clustering key to make a unique primary key. The clustering key is always a subset of the primary key.

Although SYSKEY is appended to the logical clustering key, the SYSKEY column is physically created as the first column of the table. The SYSKEY column has the data type LARGEINT.

Clustering keys apply to key-sequenced tables only.

A clustering key is useful for cases in which a table does not naturally contain a unique SQL primary key, but ordering and scanning by SYSKEY alone or access through an index is not desirable. Within the file, records can then be physically ordered by the nonunique clustering key. Using these nonunique key values, a user can scan the file more efficiently and can also create partitions.

The combined length of the clustering key, not including the appended SYSKEY column, cannot exceed 247 bytes. Columns in the clustering key definition cannot be updated.

Clustering keys have the same performance implications as primary keys, as described next under Key Levels.

Key Levels

SQL/MP keys (including index keys) can be classified in three levels. The three levels are associated with three different levels of performance related to the overhead associated with the use of the keys, as follows:

- Level 1 keys have the best performance because these keys have the least overhead. Level 1 keys have these column characteristics:
    - Columns that are contiguous, have ascending values belonging to the ASCII collating sequence, and are stored at fixed offsets
    - Combined columns defined with numeric or character data types
    - Columns with an UNSIGNED numeric data type (DECIMAL, NUMERIC, SMALLINT, INTEGER, PIC 9 COMPUTATIONAL, PIC 9 DISPLAY, DATE, TIME, or DATETIME)
    - Columns defined as having an ASCII data type (CHARACTER, PIC X, DECIMAL, or PIC 9 DISPLAY)

  For example, keys defined with these columns are level 1 (ASCENDING is the default):

```
KEY-1  NUMERIC (4.2) UNSIGNED, SMALLINT UNSIGNED
KEY-2  CHARACTER (8), DECIMAL(8) UNSIGNED
KEY-3  SMALLINT UNSIGNED, CHARACTER (24)
KEY-4  CHARACTER (24), DATETIME
```

- Level 2 keys have performance comparable to those of level 1, but with a small amount of additional overhead. Level 2 keys have these column characteristics:
    - Columns that are contiguous, either all ascending or all descending, and with a fixed offset
    - Columns with a SIGNED numeric data type (DECIMAL, NUMERIC, INTEGER, SMALLINT, LARGEINT, PIC S9 COMPUTATIONAL, PIC S9 DISPLAY, DATETIME, INTERVAL, or FLOAT)
    - Contiguous columns that are defined as numeric data but with mixed signed and unsigned columns
    - Combined columns that include at most one VARCHAR column, which cannot be the first column of the key

For example, keys defined with these columns are level 2 (SIGNED is the default):

```
KEY-1  INTEGER
KEY-2  CHARACTER (30) DESCENDING
KEY-3  CHARACTER(2), NUMERIC(8), DECIMAL(4), VARCHAR(20)
KEY-4  INTEGER, DATETIME, INTERVAL
```

- Level 3 keys include other possible keys not included in levels 1 or 2. Level 3 keys have the poorest performance because the overhead for handling these keys by the primary disk process is approximately 10 to 20 percent greater than that of level 2 keys. (Note, however, that overall transaction overhead, or response time, does not increase by 10 to 20 percent.) Level 3 keys have these column characteristics:

  - Noncontiguous columns
  - A leading VARCHAR column, which causes a nonfixed initial offset
  - Multiple VARCHAR columns
  - Columns with ASCII data but with mixed ascending and descending orders
  - Columns that use collations

  For example, keys defined with these columns are level 3:

```
KEY-1  DECIMAL (8) ASCENDING, DECIMAL (8) DESCENDING
KEY-2  VARCHAR (20)
KEY-3  Column A, D, F        <--Indicates that the columns
                                are not contiguous in the table
```

  The performance of level 3 keys is the poorest because the overhead for handling these keys by the primary disk process is approximately 10 to 20 percent greater than that of level 2 keys. (Note that overall transaction overhead, or response time, does not increase by 10 to 20 percent.)

Note. As an exception to the preceding key-level guidelines, a key column that can be defined with either numeric or character data type should typically be defined as numeric. This approach is especially important if the column is used in a predicate. The numeric data type helps achieve optimum performance, because SQL/MP computes the selectivity for numeric columns more efficiently than it does for character columns.

As an example, suppose that you want to use a key column that contains a telephone number. Because this column is not used in calculations, you can create it as either a numeric or character column; for best performance, you should choose a numeric column.

In a key-sequenced table, the data types of the columns in the user-defined primary key or clustering key (a subset of the primary key; part of a combination key) define the key level. Different key levels require increasing system overhead for processing; therefore, the keys affect performance.

The key levels and performance implications also apply to indexes, and indexes can be created on tables of all three organizations. After being created, the index is like a key-sequenced table in which the index columns make up the primary key. By using index keys to support a unique index, you can improve performance for queries.

For key level consideration, indexes always have contiguous columns because the order of the columns in the CREATE INDEX statement defining the index applies rather than the position of the columns in the underlying table. For indexes, the columns of the primary key or clustering key of the underlying table are included in the index key. For information about index keys, see the *SQL/MP Reference Manual*.

Key-Sequenced File Structure

Key-sequenced files store variable-length rows (records) that contain a primary key. New rows are stored in sequence by primary key value. A user performing update operations can update or delete rows and can lengthen or shorten values in a varying-length column (VARCHAR, NCHAR VARYING), provided the column is not part of a primary key or clustering key.

Rows are stored in a key-sequenced file logically in ascending or descending order, according to their primary-key values in conjunction with the ASCENDING or DESCENDING specification in the CREATE TABLE statement.

Defining a Primary Key

Each key-sequenced file can have only one primary key: a user-defined primary key, a system-generated SYSKEY column, or a user-defined clustering key concatenated with a system-generated SYSKEY column.

You typically create key-sequenced tables with user-defined keys, rather than using the system-defined SYSKEY. To specify a unique user-defined primary key, indicate particular columns with the PRIMARY KEY clause. Alternatively, you can specify a nonunique clustering key with the CLUSTERING KEY clause. With a primary or clustering key, you can achieve a naturally sorted order for the table that accommodates the most commonly used path to the data.

Before deciding which type of primary key to use, carefully consider the type and usefulness of the naturally sorted order of the base table, including points about key types:

- At most, only one primary key or one clustering key can be defined for a particular table; a table cannot have both a primary key and a clustering key. The key can consist of more than one column, not necessarily adjacent to each other.
- A table with a primary key or a clustering key can be partitioned.
- The columns used as a primary key or clustering key cannot be defined to allow null values. If you do not specify NOT NULL for a column in the PRIMARY KEY or CLUSTERING KEY definition, the RDBMS defines the column to exclude null values.
- A value cannot be supplied for the SYSKEY column in an INSERT statement. This rule applies to key-sequenced tables using either a SYSKEY column by itself or

SYSKEY appended to a clustering key. This rule also applies to entry-sequenced tables but not to relative tables.

- If you create a table with a SYSKEY column, consistency problems can result if the values of SYSKEY are used in a column of a user-defined table as a foreign key in that table. If the original table is reloaded, the SYSKEY values change because the column is system defined. In this event, any references to these keys in other tables will no longer point to the correct rows. When referring to keys in other tables, or for any cases in which a unique key is necessary, always use a user-defined primary key to avoid dependence upon the value of SYSKEY.
- SQL/MP primary keys, clustering keys, and alternate keys (indexes) can be composed of one or more columns of a base table. Each column of a key can be in either ascending or descending order and can be of a different data type from the other key columns. Also, the columns need not be contiguous.
- A user-defined primary key or clustering key facilitates generic locking. Generic locking enables an application to control the granularity of locks. If you define an appropriate primary key, your applications can use generic locking to improve performance.

When defining columns for a primary key, determine whether a specific set of columns is the most appropriate based on actual use of the table. If a column is used as the main access path to a table, consider defining it as the primary key or adding it to the existing primary key as the leftmost column. To determine the main access path, review transactions that use different access paths to the table.

For example, if an employee table has employee number as its primary key but most accesses are by employee name, consider defining employee name as the primary key. The key must include enough information to make it unique (perhaps by including employee number as the second part of the key). A unique alternate index can help enforce integrity in this instance.

If you change a column to be the primary key, you can change the former primary key to an alternate unique index. If the new primary key column is not unique, add a second column that makes it unique or define it as a clustering key that will have a system-defined unique column associated with it.

Types of Access

You can access key-sequenced files either sequentially or randomly. Sequential access is preferable, for example, when generating a report of the currently available quantity of all parts in an inventory file. Random access is preferable when you want to identify the vendor of a particular part.

When SQL reads a key-sequenced file by primary key, each read operation retrieves the record containing the next sequentially higher primary-key value. Similarly, when SQL reads by a clustering key, each operation retrieves the record containing the next sequentially higher value in the specified clustering-key field. When SQL reads the file through an index, each operation randomly accesses the data file.

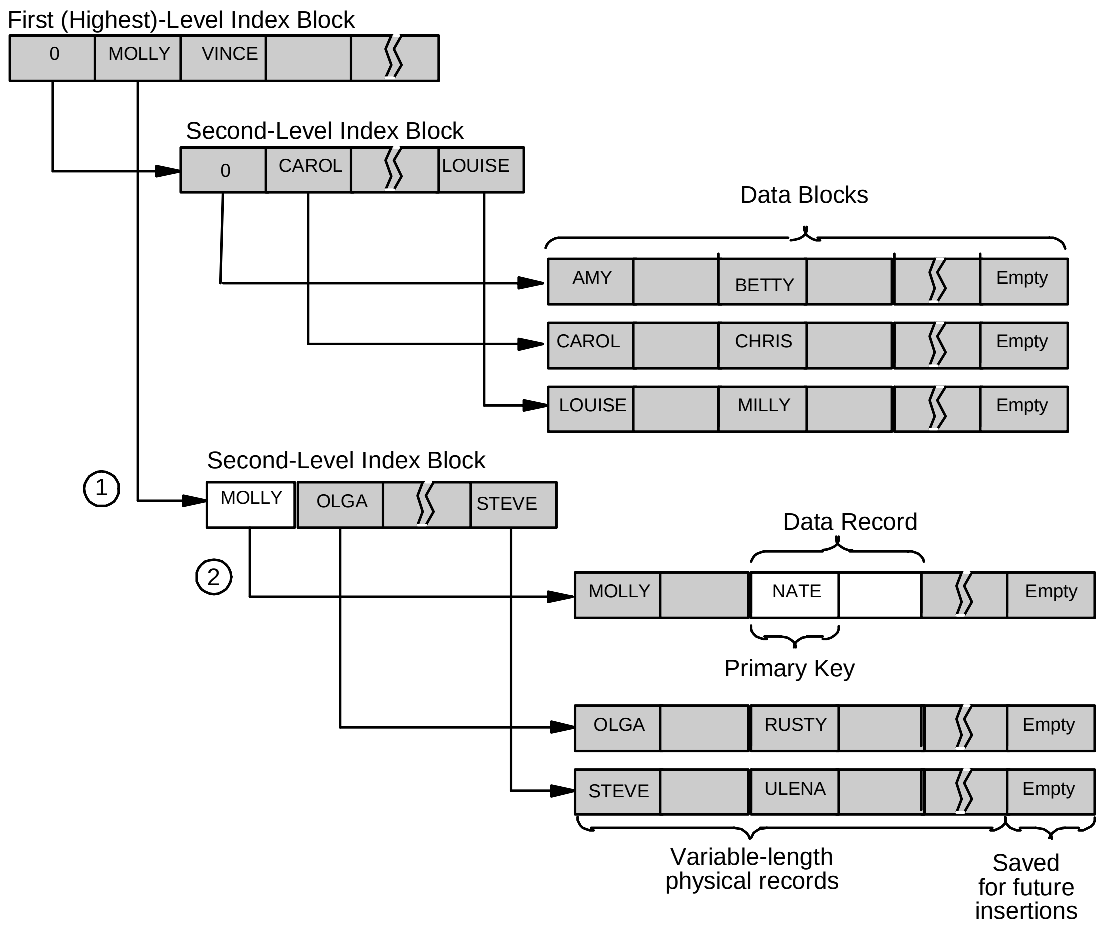
If you do not use an index, access occurs by primary or clustering key. Access can begin with the first record in the file or can requested for only a specified range of records in the file.

Key-Sequenced Tree Structure

Key-sequenced files are physically organized as one or more bit-map blocks and a B-tree structure of index blocks and data blocks. Bit-map blocks within a structured file organize the file's free space.

Figure 3-1 on page 3-8 illustrates a sample tree structure for a key-sequenced file.

Figure 3-1. Key-Sequenced B-Tree Structure



Legend

(1) NATE is alphabetically greater than MOLLY but less than VINCE. Go to the second-level index block that begins with MOLLY.

(2) NATE is alphabetically greater than MOLLY but less than OLGA. Go to the data block that begins with MOLLY.

VST002.vsd

Each data block contains a header, plus one or more data records, depending on the record size and data-block size. For each data block, an entry in an index block contains the value of the key field for the first record in the data block and the address of that data block.

The position of a new record inserted into a key-sequenced file is determined by the value of its primary-key field. If the data block is determined to be full when SQL/MP attempts to insert a new record into it, a block split occurs: the disk process allocates a new data block, moves part of the data from the old block into the new block if not at the end of the file, and gives the index block a pointer to the new data block.

Key-sequenced files are also used to store indexes. When an index block fills up, it is split in a similar manner: a new index block is allocated, and some of the pointers are moved from the old index block to the new one. The first time a split occurs in a file, the disk process must generate a new level of indexes. The disk process does this by allocating a higher-level index block containing the low key and a pointer to the two or more lower-level index blocks, which in turn point to many data blocks. The disk process must do this again each time the highest-level block is split.

The disk process can perform a three-way block split, when appropriate, creating two new blocks and distributing the original block’s data and pointers plus the new record or pointer among all three.

In a key-sequenced file, data blocks are chained together with forward and backward pointers stored in the header of each block. Data records within each block are stored in key-value sequence. This storage arrangement enhances the sequential performance for tables stored in key-sequenced files. Index blocks, however, are not chained.

Uses of Key-Sequenced Tables

A good example of the use of key-sequenced files in an application environment is an inventory file in which each record describes a part. The primary key field for that file would probably be the part number, and the file would be ordered by part number. The part numbers should be readily distinguishable from one another. Other fields in the record would contain such information as vendor name, quantity on hand, and so forth, and one or more of these fields could be used as index keys.

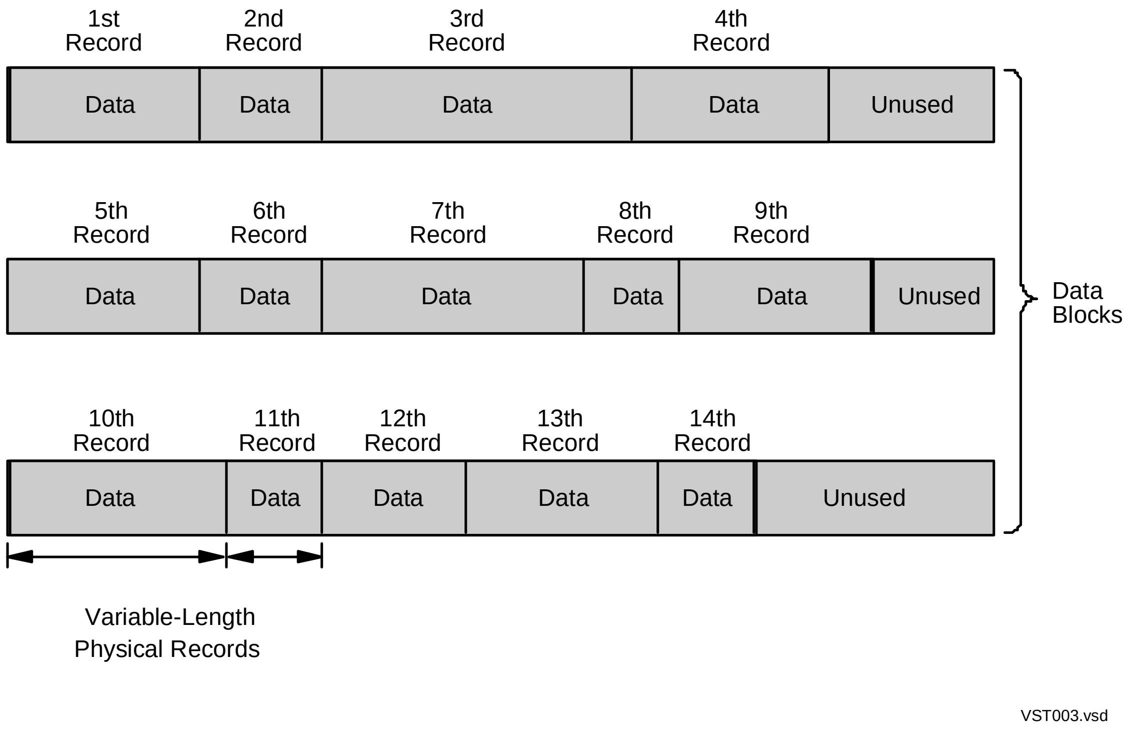
Another example of key-sequenced file use is the storage of indexes. Each index is stored in a separate SQL index file and is named after the file. An index file has a primary key that includes the indexed columns and, for nonunique indexes, the columns of the primary key of the table being referenced.

Entry-Sequenced File Structure

Entry-sequenced files are designed for sequential access. They consist of variable-length records. New records are always appended to the end of the file; as a result, the records in the file are arranged physically in the order in which they were added to the file. Users can update existing records but cannot delete them. Update operations, however, cannot increase or decrease the lengths of existing records.

Figure 3-2 on page 3-10 illustrates the structure of an entry-sequenced file.

Figure 3-2. Entry-Sequenced File Structure



The primary key of an entry-sequenced file is a four-byte record address external to the data record and consisting of a block number and a record number within the block. This address is typically used and manipulated internally by the file system, and there is usually no reason for you to know its value. Programs can, however, obtain the address of the record just read or written by calling the FILE_GETINFO[LIST][BYNAME]_ file-system procedure (for a D-series version of the Guardian operating system), or the FILEINFO or FILERECINFO file-system procedure (for a C-series version of the Guardian operating system).

After creating the file, you can perform four types of operations on the file:

- Insert new records at the end of the file
- Retrieve records from the file
- Specify an alternate access path through an index and then retrieve records that contain the key values specified by the index
- Update records (rows) of the file, without increasing the record length

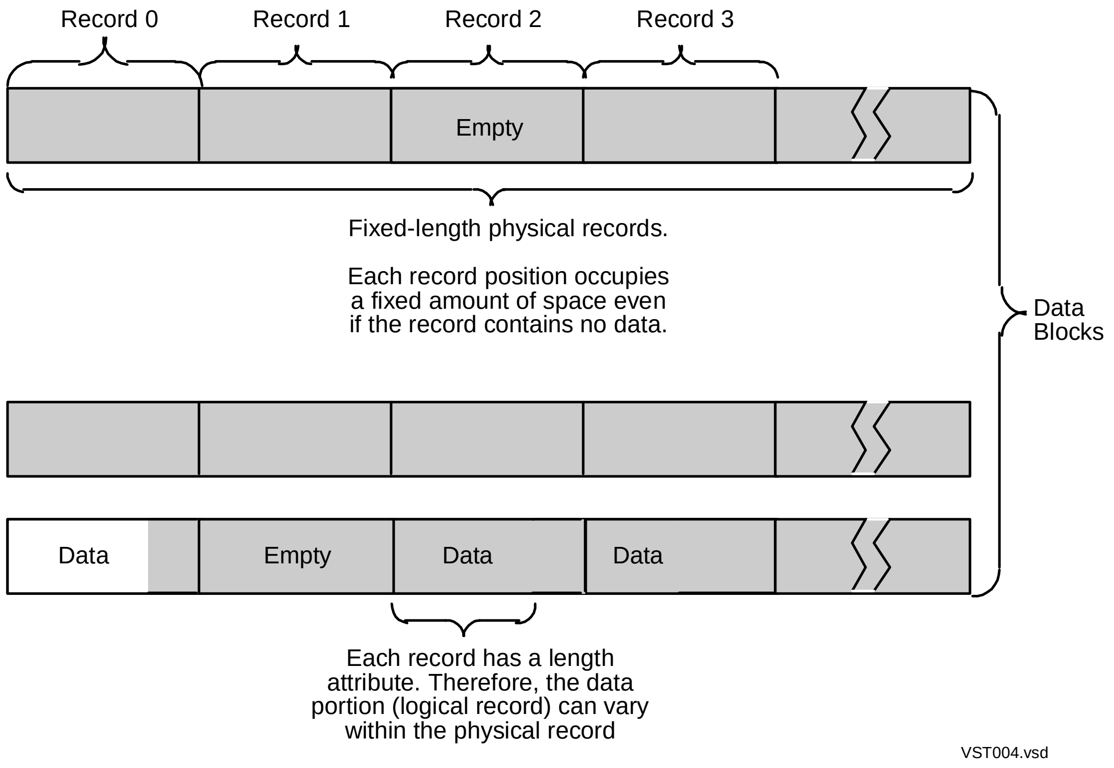
Relative File Structure

Relative files consist of fixed-length physical records accessed by relative record number. A record number is an ordinal value and corresponds directly to the record’s position in the file. The first record is identified by record number zero. Succeeding records are identified by ascending record numbers in increments of one.

The value of the primary key is either a user-specified or system-generated relative record number. Each new row is stored at the relative record number specified by the primary key.

Figure 3-3 illustrates the structure of a relative file.

Figure 3-3. Relative File Structure


Record 0
Record 1
Record 2
Record 3
Empty
Fixed-length physical records.
Each record position occupies a fixed amount of space even if the record contains no data.
Data Blocks
Data
Empty
Data
Data
Each record has a length attribute. Therefore, the data portion (logical record) can vary within the physical record
VST004.vsd

Each physical record position in a relative file occupies a fixed amount of space, and each position can contain one variable-length data record (logical record). A logical record can vary in size from zero, an empty record, to the maximum record size specified when the file was created.

You can change a record’s logical length after it has been written to the file, but the lengths of all logical records in the file must always be less than or equal to the constant size of the physical record. Each logical record has a length attribute that can be returned when a record is read. Logical records in a relative file can be logically deleted, resulting in logical records of length zero.

After you have created the file and written a data record to it, all physical records preceding that record are also created and actually occupy space on the disk, although they contain no data. For example, if you create a relative file and then write a data record to record number 135, records 0 through 134 are also physically created on the disk at that time, although they have a logical record length of zero. Note that this characteristic represents a factor that could influence both space and time performance of a relative file.

A user performing update operations on a table stored in a relative file can update or delete rows and can lengthen or shorten values in varying-length columns up to the actual defined record length.

Positioning in Relative Files

You can specify the exact position where a new row (record) is to be inserted into a relative file by using the *`system-key`* parameter in the INSERT statement. The system software accesses this position through a record number. Alternatively, you can specify one of these options:

- Insert records into any available position (ANYWHERE)
- Append records to the end of a file (APPEND)

For example, in a relative file in which only record number 10 contains data, an application can insert a new record into any zero-length record (such as record number 5). If the application indicates that any available position is permitted, the record is written to an empty location based on last file access point.

If you use the ANYWHERE option for a partitioned file, SQL/MP sets the initial “insert anywhere” partition to partition 0 when the file is opened. If an insert to that partition fails, SQL searches forward sequentially through partitions until it finds an empty position for the record. It then sets the “insert anywhere” partition to partition 0 and uses that partition for subsequent INSERT ANYWHERE operations. When that partition is full, SQL continues to search forward sequentially until an empty position is found. When the last partition in the file is determined to be full, SQL sets the “insert anywhere” partition back to partition 0. When all partitions are full, SQL returns a file full error.

Uses of Relative Files

Relative files are best suited for applications in which random access to fixed-length records is required and in which the record number has some meaningful relationship to a particular piece of data within each logical record. An inventory file, for example, could be a relative file with the part number serving as the record number. Such use, however, would probably waste disk space, because part-numbering schemes often leave large gaps in the overall number sequence; as a result, many records can be allocated but not used.

An invoice file with the invoice number serving as the record number might be a better candidate for the relative file type because there are typically no large gaps in that type of numbering scheme. If your invoice numbers begin at some large number, such as 10,000, you probably should use an address conversion algorithm to generate a record number sequence that begins at zero and then include the actual invoice number as a data field within the record.

Determining a Database Layout

In your database scheme, users and applications can access the database with these:

- Base tables only
- Views only
- A combination of base tables and views

In addition, indexes can be an efficient underlying mechanism for data access.

Using Base Tables

You can use base tables to externalize all the data to the user. Using base tables is the most direct method because views, constraints, and indexes ultimately depend on their underlying tables.

Advantages to using only base tables as the external database scheme include these:

- Security schemes and access control are simplified.
- All other SQL objects (views, indexes, constraints, and programs) are directly or indirectly dependent on the base tables.
- Recovery and management methods are simplified.

Disadvantages to using only base tables as the external database scheme include these:

- A user who has read access to a table can read any data in that table. Column masking can be done only at the application level because SQLCI queries are not restricted in reading data.
- Alterations to tables generally have a greater impact on application code that uses the tables directly. For example, an application should not use SELECT * to retrieve values from a table, because adding a column to the table would mean that the application must change.

Using Views

A view is a logical table derived from one or more base tables. A view can include a projection of columns and a selection of rows from the table or tables. You can project columns and select rows for a view directly from the base tables or indirectly through other views. You can create more than one view on a table or combination of tables.

You can use views to externalize some or all the data to the user.

A view can have the same structure as an underlying base table, or the view can be different. Views do not store data physically on the disk.

In general, views have these advantages and uses:

- Reducing the overhead of returning unnecessary rows or columns, depending on the selection criteria
- Allowing the apparent structure of retrieved data to be different from the actual stored data
- Giving individual windows on the data to many users
- Allowing logical renaming of columns
- Redefining headings for columns
- Redefining help text for columns
- Presenting only the columns and rows a user must work with, instead of all the columns and rows in the underlying table or tables

Note. A view does not insulate the programs that use it from being invalidated because of changes to the base table definition. Also, changes that do not directly affect the view require that programs be explicitly SQL compiled to be revalidated. The requirement for explicit compilation is the same for the programs used by the table directly.

SQL/MP has two types of views: protection views and shorthand views.

Protection Views

A protection view is derived from a single table by taking a projection of the columns of the table or a selection of the rows of the table, or both, and defining the view with the PROTECTION attribute. Users can change the data in the underlying table through a protection view if the view is updatable.

A protection view is updatable if the view includes all user-defined primary-key columns of the underlying table, specified in a PRIMARY or CLUSTERING KEY clause, and all columns are defined with NO DEFAULT. A protection view can be secured for read, write, execute, and purge access.

Protection views provide several features that ensure the consistency of the data:

- Protection against inserting rows that omit values for required columns

  Rows of a protection view must include values for all the columns in the underlying table that are defined with the NO DEFAULT option. If you violate this condition, an SQL warning (4056) is issued when the view is created, and you cannot insert rows into the view.

- Protection against unauthorized access to the data

  Only the local owner of a table or remote owner with authority to purge the table can create a protection view on the table. Only the owner of the underlying table can own the view.

- Limitations on access to the data

  The SECURE clause can assign a security string to a protection view to limit access to those users who have authority to read, write to, and purge the view.

- Protection against inserting or updating rows outside the definition of the view

  A protection view defined with the WITH CHECK option specifies that only rows that satisfy the view's definition can be inserted by users. Omitting this option allows rows to be inserted without satisfying the view's definition.

  Uses for protection views include:

  - Providing validity checks on the underlying table for inserts and updates
  - Providing security restrictions so that only certain information can be presented to a user by masking rows and columns of the underlying table from displays or updating
  - Masking logically deleted and added columns of the underlying table

Shorthand Views

A shorthand view is derived from one or more tables or other views and defined without the FOR PROTECTION option of the CREATE TABLE statement. A shorthand view can be read but not updated; it can be secured only for purge authority. Any user who has authority to read all tables underlying the view has authority to read the view.

One use for shorthand views is to provide security restrictions so that only certain information can be presented to the user, for display only, by specifying a set of columns and restricting rows to a given set of criteria.

When considering shorthand views for securing underlying tables from access by users of shorthand views, consider these:

- Shorthand views do not limit a user's access for reading an underlying table if the user can find the table. The security of a shorthand view depends on the underlying table. If the user is authorized to read the view, the user is authorized to read the underlying table.
- Shorthand views are difficult to secure because only the purge attribute of the security string has meaning. The other security attributes of a shorthand view are the same as for the underlying table or tables.
- A shorthand view limits a user's access to data if the user knows only about the view and not about the underlying table or tables. You can try to prevent users from reading a table by not making the name of the table available; however, a knowledgeable user could query the catalog to determine the name of the underlying table if the user has the authority to read the catalog.

Using Only Views to Externalize Data

Experienced database administrators have observed that having applications use only views as the external interface for the database scheme has certain advantages for both programs and ad hoc queries.

Advantages to using only views as the external database scheme include:

- The physical database structure of the base tables is not externalized to programmers or other users.
- The database file layout can be normalized independent of a program or of a user's interpretation.
- New base tables and views can be easily created and integrated with the existing scheme.
- Ad hoc queries are limited to the data returned by the views. Allowing users to use only views can prevent them from accessing sensitive data.
- Protection views provide the same performance as the base tables.
- Protection views can help provide data integrity.

Disadvantages to using only views as the external database scheme include:

- Backing up views for recovery involves a slightly more complex recovery strategy to ensure that you save all the view definitions. Views, however, are easily re-created because they contain no data.
- Managing the database is more difficult because of the greater number of objects.

To use only views, you should create the views as:

- For each base table, create a read-only protection view that includes all columns and rows of the table.
- For each write operation on a base table or tables, create a protection view.
- Create shorthand views based on the protection views instead of on base tables. These shorthand views can be used only for queries.

Determining When to Use Indexes

Indexes are usually used to improve performance. An index is an alternate access path to a table, which differs from the inherent access path (primary key) or clustering key defined for the table at creation. Indexes provide alternate-key sequences for files of any structure: key-sequenced, relative, or entry-sequenced. In general, an index improves access speed when the data is requested in the order of the index key.

When compiling a statement, SQL/MP selects the query execution plan for a statement by choosing the best access path to the data. If an index exists, SQL evaluates using the index. Indexes give the optimizer more possible access options.

Each index is assigned a name and is physically stored in a separate key-sequenced file that possesses the same file name as the index. Index files are not tables, and they cannot be queried directly through SQL; they are only a tool for providing faster access to tables.

Performance Benefits of Indexes

Indexes can improve performance by eliminating the need for the disk process to access the underlying table. If the query can be satisfied by the columns contained in the index and the access returns unique rows, the underlying table will not be accessed. By using only the index, you reduce I/O to the table.

For example, consider this query in which ATABLE has a unique index named AINDEX, which contains columns A and B, in that order, from ATABLE:

```
SELECT A,B FROM ATABLE
WHERE A > 100
ORDER BY A,B;
```

The query can be satisfied by accessing only AINDEX, which contains all the columns requested. This type of index-only retrieval can be effective on both unique and nonunique indexes.

Another use of an index is to eliminate a run time sort of data columns by providing an access path to the data in the order of the desired selection.

A third use of an index is to avoid a full table scan. Consider this query:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVNTRY
WHERE RETAIL_PRICE = 100
```

Without an index on RETAIL_PRICE, SQL must scan the table and evaluate the following predicate against each of the rows in the table; an index on RETAIL_PRICE would improve query performance dramatically.

An index on RETAIL_PRICE might not help this query that contains an inequality predicate, however, because ITEM_NAME is not part of the index:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVNTRY
WHERE RETAIL_PRICE > 100
```

For every index row that satisfies the predicate, an I/O operation (a request from the file system to the disk process) must be incurred to retrieve the column ITEM_NAME from the base table.

If the query selects only columns included in the index, the index on RETAIL_PRICE can help performance. Including ITEM_NAME in the index could cause index-only access, which would improve the performance of this query.

Evaluating the Benefit of a New Index

Creating alternate indexes can help the performance of some, but not all, queries. In some applications, determining when an index could be efficient might be easier than in others. Examine the WHERE clauses and ORDER BY clauses of the SELECT statements in your application. You should consider creating indexes on only the most frequently used columns.

In general, indexes are an efficient way to access data if the following are true:

- The number of rows retrieved is small.
- The result is presented in a certain order or grouped according to certain columns, such as queries that use a DISTINCT, GROUP BY, ORDER BY, or UNION clause, which can cause a sort operation if indexes are not available.
- An application has many queries that refer to a column in a table.
- All the necessary information can be obtained from the index (index-only access).
- The column is an argument of the MIN or MAX function.

Columns used as selection criteria, but that are also frequently updated, might not improve overall performance as an index. When a column that is also an index column is updated, both the table and index require updating. The system automatically updates the index when it updates the table. Updating the index slightly degrades performance for the update operation. The index, however, might improve performance for the selection operation.

The use of an alternate index does not ensure that the optimizer will choose the index as the access path. Index use depends highly on selectivity, described in the *SQL/MP Query Guide*. In general, these guidelines apply:

- If the query can be satisfied with an index-only access, the optimizer uses the index.
- For base-table access through an index, the optimizer performs a random access read against the base table. If index selectivity becomes too high, the optimizer scans the base table instead of using the index.

You can use the EXPLAIN facility (described in the *SQL Query Guide*) to determine if the extra index will be used by SQL for a particular query. Furthermore, each additional index adds overhead during updates. Queries that update index columns incur the overhead of having one more index to update.

Note. After you create an index for a table, run UPDATE STATISTICS. Otherwise, SQL returns a warning for subsequent operations that access that table.

Defining an Index

When you define an index, consider these guidelines:

- The maximum length for the rows of a nonunique index is 253 bytes. The row length includes the sum of the lengths of the columns declared for the index plus the sum of the lengths of the columns of the primary key of the underlying table.
- The maximum length for the rows of a unique index is 508 bytes. The rows can include 253 bytes for the KEYTAG column and indexed columns and 255 bytes for the primary key of the underlying table.
- For varying-length columns (VARCHAR, NCHAR VARYING), the length referred to in these limits is the defined column length, not the stored length. The stored length includes two additional bytes in which the RDBMS records the data length of the item. For example, if the index includes VARCHAR columns, the actual stored record length would be two bytes greater for each VARCHAR column than the defined column length.
- If there are ordering requirements, consider defining the sequence of columns so that it meets those requirements. Otherwise, a sort will be necessary to fulfill the ordering requirements.
- If an index is unique, define it as unique. SQL can access the index more efficiently if the index is unique and specify equality predicates on all index columns.
- In general, do not explicitly include primary key columns in an alternate index. These columns are already stored at the end of the index. However, if primary key columns are used for positioning or in an ORDER BY clause, consider including those columns as part of the alternate index. This approach might avoid a sort operation.
- If you frequently access a set of columns that is almost contained within your index, consider adding the remaining columns to the alternate index to create index-only access for such queries. This approach increases storage requirements and update processing of those columns, so you should evaluate these trade-offs.
- If you access a set of information—the same values in several rows, such as all names equal to Smith—consider using primary-key access for that data instead of alternate index access.

Defining the Key for an Index

The primary key for an index file includes these columns:

- KEYTAG—a unique identifier for an index on a base table. KEYTAG is a two-byte column that can contain either two characters or data of type SMALLINT UNSIGNED with values from 1 through 65,535. All rows of a given index have the same KEYTAG value. The KEYTAG values can either be user specified or system generated, but each value must be unique among the set of KEYTAGS defined on the table. If SQL generates KEYTAG values, it sequentially numbers all indexes on a table, beginning with 1. KEYTAG 0 is the primary key.

- Indexed columns—the columns located in the column list in the CREATE INDEX statement.
- For nonunique indexes—columns of the primary key of the underlying table. These columns are required to identify rows uniquely in the index. The primary key of a nonunique index automatically includes the columns of the primary key of the underlying table to associate the indexed columns with the rows of the table. In a unique index, the columns of the primary key of the underlying table are not logically included in the primary key of the index, but are physically included in the index file.

The primary key of an index differs from the primary key of a table because primary key columns of an index can contain null values. In calculating the length of an index key, you must include the null indicator (two bytes) in the length for each column that allows null values.

Creating Indexes for Specific Situations

These subsections describe the use of indexes for specific situations.

Creating an Index for Frequently Used Columns

If an application has many queries that refer to a column in a table, an index on that column might improve the performance of some of the queries. For example, consider this query:

```
SELECT QTY_ORDERED, RETAIL_PRICE FROM INVNTRY
   WHERE RETAIL_PRICE = 100 ;
```

If there is no index on RETAIL_PRICE, SQL must scan the table and evaluate the predicate (RETAIL_PRICE = 100) against each of the rows in the table.

An index on both columns would enable an index-only access; that is, SQL could retrieve all required data from the index and not have to read the base table. Consider this query, for example:

```
CREATE INDEX RPRICE
   ON INVNTRY (RETAIL_PRICE, QTY_ORDERED) ;
```

Note. If a column such as QTY_ORDERED is frequently updated, an index on the column incurs the index maintenance cost resulting from several insert and delete operations. This approach might reduce the benefit obtained from having the index defined on the table.

For another example, suppose that applications frequently access the EMPLOYEE table by employee name (and the primary key is EMPNUM). You can speed the execution of these queries by creating an index of employee names based on the LAST_NAME and FIRST_NAME columns, as shown in this example:

```
CREATE INDEX XEMPNAME
   ON EMPLOYEE (LAST_NAME, FIRST_NAME) ;
```

Avoiding Sort Operations

Indexes can help the performance of queries that might require a sort operation, such as queries that contain one or more of these keywords:

DISTINCT

GROUP BY

ORDER BY

Consider this query:

```
SELECT * FROM INVNTRY
  ORDER BY QTY_ORDERED, RETAIL_PRICE ;
```

If the INVNTRY table is large, the cost of sorting the table might be very high. An index on the columns QTY_ORDERED and RETAIL_PRICE, defined as follows, might mean that no sort is required to satisfy the ORDER BY clause:

```
CREATE INDEX RPRICE
   ON INVNTRY (QTY_ORDERED, RETAIL_PRICE) ;
```

A large cache size would also help ensure the efficiency of such a query.

To avoid a sort, define an index that has the same key columns as the sort key columns; the sequence of these columns in the ORDER BY clause should then match the sequence of columns in the index. Ordering requirements should be explicitly stated, however.

Do not assume that rows will be returned in a specific order because of the primary-key sequence. Selectivity considerations might cause the optimizer to choose an alternate index, and the rows might not be in the desired primary-key sequence. For more information, see the *SQL/MP Query Guide*.

If the ORDER BY clause is specified for a nonkey column, consider adding the column to the index, right after the matching index columns. Subsequent ORDER BY operations would then refer to all preceding matching columns plus these ordering columns. If the order of key or index columns changes in the database, notify users and programmers so that they can change ORDER BY clauses to match the new sequence.

Creating Indexes for MIN and MAX Functions

Indexes can also improve the processing of the MIN and MAX functions. For example, consider these two queries.

Suppose that an index exists on the RETAIL_PRICE column. The same index can be used to read both the MIN and the MAX example:

```
SELECT MIN(RETAIL_PRICE) FROM INVNTRY ;
SELECT MAX(RETAIL_PRICE) FROM INVNTRY ;
```

The first query can be evaluated by reading a single row from the index (specifying a forward read) to satisfy MIN (RETAIL_PRICE). The second query can be evaluated by reading a single row from the index (specifying a backward read) to satisfy MAX (RETAIL_PRICE).

Note. If you specify both the MIN and MAX functions in a single query, a scan of the index is necessary.

Creating Indexes to Improve OR Operations

Indexes help improve the performance of the OR operator if the predicates involve reference index columns.

The term “key prefix” refers to a set of contiguous key columns taken from the leftmost key column onwards. For example, if an index I contains 3 key columns (A, B, C), then there are three key prefixes: A, AB, and ABC. The prefix ABC corresponds to the full key, the other prefixes form a partial key.

In this example, COL1 and COL2 are key prefixes from two different indexes. SQL uses the indexes to retrieve all the rows that satisfy the predicate COL2 = 20 and do not have COL1 = 10:

```
SELECT * FROM T
   WHERE COL1 = 10 OR COL2 = 20 ;
```

Evaluating the query by using the indexes is much more efficient than scanning the entire table.

4 Planning Database Security and Recovery

Database security and recovery are two important topics to consider before creating a database:

- Planning for security and implementing an authorization scheme is the primary protection against unauthorized user intervention. Security, however, cannot eliminate errors by authorized users.
- Planning for recovery is essential for protecting your database. Your recovery plan should include protection against disk failures, software failures, application errors, other equipment failures, catastrophic disasters, and human errors of all types.

HP NonStop software provides several online recovery mechanisms, including:

- Mirrored disk volumes are a primary protection against disk failures. These volumes also provide the ability to repair and maintain disk volumes online, without interrupting application processing.
- The TMF subsystem provides the best online protection against application or equipment failures. When used correctly, the TMF subsystem protects the database from program failures that would leave the database inconsistent because of incomplete transactions.
- The use of backup tapes of the data files can provide a way of protecting data in an offline mode. Tapes can be physically removed from the site and saved for possible disaster recovery.
- The Remote Duplicate Database Facility (RDF) maintains replicated databases at a remote site that can be used for contingency planning. As end users modify the local database, RDF replicates those changes in the remote database, keeping it continuously up to date. For more information about managing a replicated SQL/MP database with RDF, see the *RDF/IMP and IMPX System Management Manual.*

NonStop fault-tolerant hardware and software strategies provide maximum protection against most equipment failures, power failures, and some catastrophic failures. This protection, however, does not eliminate the need to plan carefully to protect your database and application software. After formulating a comprehensive recovery strategy, practice carrying out the plan on a regular, consistent basis.

Security Guidelines

Authorization to operate on SQL tables, views, indexes, collations, and SQL programs that run in the Guardian environment is maintained by the Guardian security subsystem. Authorization for SQL programs that run in the OSS environment is maintained by the OSS security subsystem. When planning security, consider the needs or restrictions of all the users of a system, or a network of systems, in addition to the needs or restrictions of a particular database.

When planning authorization schemes, consider:

- What are the requirements for security on the system or network?
- How many different user groups use the same database?
- What are the anticipated requirements for cross security between databases or user groups?
- Which users should have the authority to change the data dictionary?
- Which users should be given authority to purge SQL objects?

This discussion on planning authorization provides examples of authorization schemes. Section 5, Creating a Database, lists security guidelines related to specific types of database objects. For more information on Guardian security, see the *Security Management Guide*.

Sample Authorization Schemes

Application needs on a system can define the needs for security authorization. Usually, authorization schemes affect the number of catalogs you choose for your system. In general, you should create the smallest number of catalogs logically possible, as dictated by your business operations.

Three examples of possible application security and catalog schemes follow.

- Production banking system

  This system has a limited number of user groups but high business activity and strong security requirements for database management operations. This scheme probably should use one, two, or just a few catalogs.

  Characteristics of the application are:

  - The production application should be valid without automatic recompilation.
  - The database should be stable because only a few changes would be made for location, security, or other DDL operations.
  - Only the database administrator or the super ID user can perform DDL operations, so that the catalogs are secured for access only by the DBA or the super ID.

    - The tables, views, and indexes are secured for access by servers; all application use is through programs initiated by the application environment.
    - Queries on tables or views are limited to the database administrator and the super ID user.

  The most important security factor in this environment is securing the catalogs from unauthorized DDL statements that could alter the database or from any operations that could allow an unauthorized program to be registered.

- Development system

  In this system, many user groups share the same or similar databases while the application passes from development to testing, to documentation, and finally to release control. This scheme probably should use one or more catalogs for each user group.

  Characteristics of system use are:

    - Each user group needs control of the database and the ability to register programs in a catalog.
    - The user groups might share a database, and changes to the catalog descriptions must be coordinated with each group.
    - If each user group uses a separate catalog, users will frequently copy tables, dependent views, and indexes by using the DUP command.

  The most important security factor in this environment is securing the catalogs and objects so that users can perform the many development tasks. The catalog and object security should be simple to allow an authorized user to duplicate the entire application for the next phase of development.

- Several unique application groups

  These user groups share a system but have unique databases. This scheme should use a system catalog plus one or more catalogs for each application group. Users do not need to move or copy objects among these catalogs.

  Characteristics of system use are:

    - Each user group has a database administrator to manage the database and the application for the group.
    - Each user group wants autonomy and protection from the other groups. The important security factor in this environment is the ability to restrict accidental use by other groups.

For authorization in general, you should create the simplest authority and security scheme possible. Dependent views, indexes, and programs should be owned by the same user ID, and only that user ID should have purge authority. With this authorization scheme, DDL operations and utility operations that can affect the entire set of dependent objects, such as DUP, are simplified. Because anyone who has authority to purge an object can drop that object, an authorization scheme should limit the authority for purging.

For an authorization scheme, you should establish catalog boundaries along the lines of application and user access requirements. Associate catalogs with sets of tables logically associated or used together. With this scheme, security follows the catalogs you choose.

Guidelines for Security Schemes

When planning a security scheme, consider these guidelines:

- Security issues closely follow the use of three categories of SQL statements. These categories and the most frequent users of each category are:
    - Data Definition Language (DDL) statements, issued by the database administrator
    - Data Control Language (DCL) statements, issued by application users
    - Data Manipulation Language (DML) statements, issued by application users
- The local owner of a table, view, index, collation, or program, the local group manager, the local super ID, or the remote owner with purge authority generally has the authority to perform DDL statements on these objects. Authority to purge an object is required to drop a table, index, view, SQL program stored in a Guardian file, or collation from the database.
- A group manager (user 255) can read or write to any local table owned by a group member and can execute an SQL program that runs in the Guardian environment that is owned by any group member. Remote tables, views, and programs must be secured for remote access. When a statement requiring access to an object is compiled, the catalog that describes the object must be accessible by the group manager. To alter attributes of a table, view, index, collation, or SQL program stored in the Guardian environment, or to run a DDL statement, a group manager requires purge authority.
- SQL/MP security issues cover two areas:
    - Security of a catalog that contains descriptions of SQL objects
    - Security of SQL objects

  Allowing access to the catalog does not automatically allow access to the objects described in that catalog. Access to the catalog is required in addition to access to the objects for execution of:
    - DDL statements
    - DML statement compilations for SQLCI or dynamic SQL
    - Most utility commands
    - SQL program compilations

Network databases require remote passwords (at the network level) and network security strings for both catalogs and objects to allow remote access.

- When an SQL object is created, the ownership defaults to the owner of the session or program. The security of the object defaults to either the security of the underlying table or the current default security, unless the statement creating the object provides another security string. Section 5, Creating a Database,contains additional object-specific information about security.
- The security attributes of a table, view, index, or SQL program that runs in the Guardian environment can be changed by an ALTER statement.
- The security string for an object must be set to allow users who have write authority to also have read authority.
- A change in the ownership of an object affects the interpretation of the security string. SQL interprets the security string at run-time against the user ID of the new owner. The change does not apply to a running SQL program until program execution ends.
- The owner and security of an underlying table determine those attribute values for indexes on the table. If you change the owner or security string for the underlying table, SQL automatically changes the owner or security string for any indexes on the table.
- The CLEARONPURGE and NOPURGEUNTIL attributes for a table do not dictate these attribute values for dependent indexes. You can set these two attributes independently for indexes.
- The owner of a base table determines the owner of a dependent protection view. If you change the owner of a table, SQL automatically changes the owner of any dependent protection view.
- If you change the owner of a program, SQL automatically sets the PROGID attribute to NO, regardless of the original setting.

Authorization Requirements for Database Operations

When creating a database, it is important to understand the authority necessary for various types of operations on tables and programs. Table 4-1 on page 4-6 describes what authority users must have to use specific statements and commands. For DDL statements, users must also have authority to read and write to any catalogs affected by the change.

Table 4-1. Authorization Requirements (page 1 of 3)

Compile and Run Commands

Command	**Authority Required**
SQLCOMP	Read and purge authority for the program file; read and write authority for the PROGRAMS, USAGES, and TRANSIDS table of the catalog in which the program will be registered; and read and write authority for the USAGES and TRANSIDS catalog tables of any catalog that contains a description of a table or view that the program uses.
Binder program	Same authority requirements as for SQLCOMP.
RUN program file	Read and execute authority for the program file; for dynamic recompilation, read authority for any catalog with a description of a table or view used by the program.

DCL Statements

Command	**Authority Required**
FREE RESOURCES	Read authority for affected objects.
LOCK TABLE UNLOCK TABLE	Read authority for the table or view and all underlying tables of the view.

DDL Statements

Statement	**Authority Required**
DDL commands in general	Read and write authority for affected catalogs unless otherwise noted.
ALTER	Local owner of the object, local super ID, local group manager, or remote owner with purge authority for the object (or for the underlying table if the object is an index).
To resecure program	Read and write authority for the affected catalog and for the program file.
To resecure catalog	Either local owner or remote owner with purge authority for the catalog.
COMMENT	Local owner of the referenced table, view, or underlying table of the index described by the comment; local super ID; local group manager; or remote owner with purge authority for the object.
CREATE CATALOG	Write authority for the SQL.CATALOGS table on the system that contains the catalog.
CREATE COLLATION	Read and write authority for the catalog in which the collation will be registered and read authority for the collation source file.
CREATE CONSTRAINT	Local owner of the underlying table, local super ID, local group manager, or remote owner with purge authority for the table and read authority for the underlying table.

Table 4-1. Authorization Requirements (page 2 of 3)

CREATE INDEX	Local owner of the underlying table, local super ID, local group manager, or remote owner with purge authority for the table; read and write authority for the underlying table; and write authority for the USAGES table of catalogs that describes the underlying table.
CREATE TABLE	Read and write authority for all affected catalogs.
CREATE VIEW Shorthand	Write authority for the USAGES and TRANSIDS tables in catalogs that describe the underlying tables and views and write authority for the VIEWS catalog table.
CREATE VIEW Protection	Local owner of the underlying table, remote owner with purge authority for the table, or the local super ID or group manager.
DROP CATALOG	Read and purge authority for the catalog and read and write authority for the SQL.CATALOGS table.
DROP CONSTRAINT DROP INDEX	Local owner, local super ID, local group manager, or remote owner with purge authority for the underlying table.
DROP PROGRAM DROP TABLE DROP VIEW	Purge authority for the object being dropped.
UPDATE STATISTICS	Local owner, local super ID, local group manager, or remote owner with purge authority for the table for which statistics are being updated.
DML Statements	
Statement	**Authority Required**

DELETE
INSERT
UPDATE
	Read and write authority for the table or protection view being deleted or modified and read authority for tables, protection views, and underlying tables of shorthand views specified in subqueries of the statement.

SELECT	Read authority for tables, protection views, and underlying tables of shorthand views specified in the statement.
Utility Commands	
Command	**Authority Required**
CLEANUP	Local super ID.
CONVERT	Read authority for the file to be converted and the DDL dictionary and the same authority as for CREATE TABLE, CREATE INDEX, and LOAD.
COPY	Read authority for the source file or object; write authority for the target file or object; and for objects, read authority for the catalogs containing the object descriptions.
DISPLAY USE OF	Read authority for the catalogs containing the object descriptions.

Table 4-1. Authorization Requirements (page 3 of 3)

DUP	Read authority for objects and files being duplicated; read authority for the catalogs containing the object descriptions; same authority as for CREATE statements for the types of objects being duplicated; and purge authority for target files and objects if purging is necessary.
EDIT	Read and write authority for the file to be edited.
FILEINFO	Read authority for each object or file for which statistics are to be displayed.
INVOKE	Read authority for the catalogs containing the object descriptions.
LOAD	Read authority for the source file or object; write authority for the target file or object; and for objects, read authority for the catalogs containing the object descriptions. If the target file is a table, then LOAD requires the authority to write to the catalog in which the table is described.
MODIFY [DICTIONARY]	Local super ID unless the CHECKONLY option is specified. For a MODIFY LABEL CHECKONLY request, read authority for the SQL objects and object programs. For a MODIFY CATALOG CHECKONLY request, read authority for the catalogs.
PURGE	Same authority as for DROP for objects being purged and local super ID, local group manager, or purge authority for files being purged.
PURGEDATA	Write authority for the files and for the tables and affected catalogs.
SECURE	Same authority as for ALTER for the object being secured and owner of the file, local group manager, or local super ID.
TEDIT	Read and write authority for the file to be edited.
UPGRADE CATALOG	Local owner of the catalog, local super ID, local group manager, or remote owner with purge authority for the catalog tables, and write authority for the system CATALOGS table.
UPGRADE SYSTEM CATALOG	Local super ID.
VERIFY	Read authority for the catalogs containing the object descriptions.

For a full explanation of the authorization scheme, see the *Guardian User’s Guide*.

Safeguard Security Product

For additional security protection, you can use the Safeguard product to restrict access to volumes and subvolumes containing SQL tables, views, indexes, collations, and SQL programs stored in Guardian files. You can use the Safeguard product to protect an entire catalog by protecting the subvolume that contains the catalog.

The Safeguard product can authorize or prevent all attempts to access protected system objects, including disk files, disk volumes and subvolumes, devices, and named processes. The owner of a system object can create an access control list that specifies the users and user groups who can or cannot access the object. If an access control list does not specify access permission for a particular user, the Safeguard product rejects that user's access attempt.

The Safeguard product has these general attributes:

- The Safeguard product can restrict the creation of tables, views, indexes, collations, and catalogs on volumes and subvolumes for which it maintains a user-authentication record.
- The Safeguard product can protect the creation of SQL processes and the execution and purging of SQL program files.
- Safeguard access lists cannot be created for individual SQL object names, although names of SQL tables, views, indexes, collations, and programs are disk file names.
- The Safeguard product works with the Guardian security system to enforce the security controls established by system managers, security administrators, and other users.

A Safeguard user-authentication record represents each user, and the owner of the record controls the security attributes for that user.

Before a volume is protected by the Safeguard product, anyone with access to the system can create objects on that disk volume.

To use Safeguard authorization control for creating SQL tables, views, or indexes, you must add the disk volumes or subvolumes on which these objects will reside to the Safeguard protection scheme. Every table for which you want a different access control list should reside on a different subvolume. For a partitioned table or index, you must secure each volume containing a partition of the object individually, providing the same Safeguard protection for each partition.

To set up a volume, subvolume, or process under Safeguard protection, you must invoke SAFECOM, the command interpreter for the Safeguard product. Then you can alter the access for the volume, subvolume, or process, as in this example:

```
SAFECOM
= ASSUME VOLUME;
= ALTER $DATA,ACCESS \*.GROUP1.USERID  C;
= ALTER $DATA,ACCESS \*.GROUP2.*  C;  ****
.
.
= ASSUME SUBVOLUME;
= ALTER $DATA.PERSNL , ACCESS \*.GROUP1.USERID  (C, P);
= ALTER $DATA.SALES  , ACCESS \*.GROUP2.*  (C, P) ;
= ALTER $DATA.INVENT , ACCESS \*.GROUP2.*  (C, P) ;
= ALTER $DATA.DPROGS , ACCESS (\*.GROUP1.*, \*.GROUP2.*)  (C, E, P)
.
.
= ASSUME PROCESS ;
= SET OWNER 100,255 ;
= SET ACCESS 100,255 (R, W, E, P, C); 200.* DENY (E, P);
.
.
= EXIT;
```

The TMF Subsystem

The TMF subsystem provides transaction protection, database consistency, and database recovery.

TMF Concepts

Use of the TMF subsystem requires an understanding of these TMF elements:

- Transactions
- Audit trails
- Audit files
- Audit dumps
- Online dumps

These paragraphs give a brief overview of these elements; for more information, see the *TMF Introduction*.

Transactions

A transaction, in general, is a multistep operation with a designated beginning and end that changes a database. For example, a transaction for an airline reservation could include the operations of adding a reservation to the airline passenger list, issuing a ticket, and adding the ticket price to the accounts receivable table. Transactions associated with SQL/MP operations are called TMF transactions.

A TMF transaction can span numerous database changes that affect multiple files on multiple disk volumes and nodes. The TMF subsystem can abort an incomplete transaction if a failure occurs during the transaction, thus ensuring consistency—either all or none of the changes in a transaction are applied to the database. During normal processing, the TMF subsystem also maintains the necessary locks on data to ensure consistency of the database.

Audit Trails

If a system, disk, or program fails during a transaction, the TMF subsystem uses audit trails to restore the files to their original state before the start of the transaction. Each audit trail is a series of files in which the TMF subsystem records information about transactions' changes to a database. The information includes:

- Before-images, which are copies of data records before a transaction changes them
- After-images, which are copies of the changed records

Audited Files

Files or tables protected by the TMF subsystem are called audited files. Only audited files have changes logged to audit trails. Files not protected by the TMF subsystem are nonaudited files and do not have changes logged. You can choose which files are to be audited on a file-by-file basis, depending on application requirements. Only files on a TMF-configured data volume can be audited.

Audit Dumps

An audit dump is a copy of an audit trail file written to a tape or disk volume by an audit dump process. If audit dumping is configured, audit dumps occur automatically when an audit trail file becomes full. An audit dump process can be configured for each audit trail; it can be reconfigured while the TMF subsystem is running. Audit dumps are used by the file recovery process; they remain either on audit-restore volumes or on the audit dump medium (disk or tape) until they are no longer needed for recovery.

Online Dumps

An online dump is a copy of an audited database file written to tape or disk in case of media failure or other damage to a database such as an accidental purge operation. Each online dump of a file provides an image of a file that can be used by the file recovery process to reconstruct the file. Thus, online dumps are essential for most file recovery operations.

An online dump is created when a TMF DUMP FILES command is issued. Online dumps can be made while transactions are being processed by database applications.

Levels of Database Recovery

The TMF subsystem provides three mechanisms for database recovery: transaction backout, volume recovery, and file recovery.

The consistency of an SQL database is ensured if any TMF recovery operation completes without errors. TMF recovery methods protect both SQL catalog tables and audited SQL objects.

Transaction Backout

Transaction backout provides automatic online recovery for aborted transactions. A transaction is aborted when an event prevents the transaction from being committed. Possible events include:

- Program suspension or abnormal termination because of an error or specific programmatic request
- Processor failure
- Communication failure between participating nodes of a network-distributed transaction

The TMF subsystem handles backout operations without operator intervention by using the audit trails automatically cycled by the TMF subsystem. The TMF backout process uses before-images in the audit trails to undo the effects of an aborted transaction.

Volume Recovery

Volume recovery recovers the database in the event of a disk crash or system failure. When the TMF subsystem is restarted after a failure, volume recovery is initiated automatically for each accessible data volume on the system (except for volumes explicitly disabled in TMF).

To recover the files, the volume recovery process re-applies committed transactions to ensure they are reflected correctly in the database, and then backs out all transactions that were incomplete at the time of the interruption.

File Recovery

File recovery reconstructs specific audited files when the current copies on the data volume are not usable: for example, if a system or media failure jeopardizes the consistency of one or more audited files. A file could require file recovery for one or more reasons, including:

- A disk failure (irreparable media failure) occurs.
- A volume or system failure occurs, and volume recovery cannot recover the file.
- A file is mistakenly purged.
- An application program incorrectly changes the database.

File recovery includes restoring online dumps from tape to disk, applying the after-images from the audit trail to the database records, and then backing out all transactions that were incomplete at the time of the system interruption or failure.

SQL Requirements for TMF

To protect the data dictionary in recovery situations, SQL/MP requires auditing of the SQL catalogs by the TMF subsystem.

(Similarly, volumes that contain SQL objects, except programs, must be enabled for auditing by the TMF subsystem.) Individual SQL tables, indexes, and views can be nonaudited; however, both audited and nonaudited objects must reside on audited volumes because the file labels are audited.

SQL catalogs, tables, views, indexes, collations, and partitions of tables and indexes must reside on volumes enabled for auditing by the TMF subsystem. (Similarly, volumes that contain SQL objects, except programs, must be enabled for auditing by the TMF subsystem.)

Individual SQL tables, indexes, and views can be nonaudited, although both audited and nonaudited objects must reside on audited volumes because the file labels are audited. Consider using the TMF subsystem to audit all tables, views, and indexes to ensure both the integrity of the database and a timely recovery from media failures or incomplete transactions.

As a general rule, the TMF subsystem must be available when users are running SQL application programs or using SQLCI. In particular, the TMF subsystem is required for these operations:

- DDL statements
- SQL compilations, whether explicit or requested interactively through SQLCI or through dynamic SQL applications
- DML statements performing INSERT, UPDATE, or DELETE operations on audited tables or views
- SELECT statements not using BROWSE ACCESS on audited tables or views

Some previously compiled programs or previously prepared DML statements, however, do run successfully when the TMF subsystem is unavailable, provided that the statements do not require TMF transactions. These DML statements include:

- Queries (SELECT statements or cursor operations) that specify BROWSE access on audited tables and views
- SELECT, INSERT, DELETE, and UPDATE statements that access only nonaudited tables and views

Nevertheless, these queries and statements also fail if automatic recompilation is required; for example, if an object in the access path becomes unavailable.

Guidelines for Configuring TMF

The appropriate version of the TMF subsystem (as described under Hardware and Software Requirements on page 2-1) must be installed, configured correctly, and active for transaction processing on a system before you install SQL/MP.

These guidelines apply to configuring the TMF subsystem for use with SQL/MP.

Determining What to Audit

You should audit all volumes on your system except the TMF audit-trail volume. Tables in which data changes, during INSERT, UPDATE, or DELETE operations, need auditing to protect the consistency of the database. This configuration enables you to place SQL objects throughout the system. Normally, the volume that contains the audit trails is not audited; therefore, SQL objects would not reside on this volume.

For certain systems with limited disk space, you can configure the TMF subsystem with SELFAUDIT, which allows the volume containing the audit trails to be audited also. With this configuration, SQL objects can reside on the same volume as the audit trails.

Certain tables should not be audited. For example, a log file should not be audited because it typically records various events. If a log file is audited, the TMF subsystem backs out event records, thereby eliminating valuable historical information about events such as failures.

A database with a combination of audited and nonaudited tables can be left in an inconsistent state after a failure. If a failure occurs, audited tables are recovered to the original state, but updates to nonaudited tables are left in an unknown state. You will need a strategy to recover the nonaudited tables so that the database will be consistent.

The default volume for the system catalog is $SYSTEM. If you intend to use this default volume, $SYSTEM must be audited.

Determining a Level of Data Protection

Configure the TMF subsystem for the level of protection your application needs. The minimum level of protection uses the automatic recovery features of TMF backout and volume recovery. Audit dumps and online dumps allowing for file recovery are optional. After you determine the level of protection you need, configure the TMF subsystem accordingly.

Size Considerations

When determining the size requirements of the TMF subsystem for SQL/MP, consider these:

- The catalogs are audited tables; therefore, insertions and updates to catalogs are audited.
- DDL statements run within system-defined TMF transactions, generating audit-trail entries. DDL statements that refer to large tables can generate a large volume of audit-trail entries.
- Transaction volume includes database use by both application programs and SQLCI interactive capabilities. The interactive volume might be minimal or might generate many audit-trail entries.
- You should estimate insert, update, and delete transaction activity for each table and view.
- If parallel update and delete operations are being done, consider increasing the amount of audit trail space available to TMF. For more information, see the *TMF Operations and Recovery Guide.*
- For Enscribe systems being converted to use SQL/MP, you might need to increase the size of TMF audit trails to accommodate the catalog auditing and database manipulation activity.
- To prevent suspension of TMF transactions, you should have at least two tape drives available. If you have only one tape drive and the drive fails, the TMF subsystem suspends all new transactions if the maximum number of files is reached.

Specifying TMF Attributes

Use these guidelines when specifying TMF attributes:

- You can define a separate audit trail for each volume or audit more than one volume in the same audit trail.
- The volume or volumes containing the audit trails must have sufficient free space to accommodate the extents required for the number of audit-trail files. If there is insufficient space to create a new audit-trail file, transactions can be suspended while the TMF operator dumps older audit-trail files to tape and frees enough space to continue.
- The amount of audit-trail data generated can vary depending on the setting of the AUDITCOMPRESS attribute for the audited database files. Using AUDITCOMPRESS saves system resources for update operations. Using NO AUDITCOMPRESS enables you to read the TMF audit-trail files with complete before and after images. AUDITCOMPRESS is the default for audited tables, including catalog tables. (A CREATE INDEX operation that uses the WITH SHARED ACCESS option always uses the NOAUDITCOMPRESS option.)
- For a protection view, the AUDIT attribute value is automatically the same as the value for the underlying table.
- For an index, the AUDIT attribute value is automatically the same as the value for the base table.
- By altering the value of the AUDIT attribute for a base table, you also alter the value for any dependent views and indexes.

For additional information on TMF configuration parameters and protection methods, see the *TMF Planning and Configuration Guide*.

Guidelines for Online Dumps

Correct handling of online dumps is essential for effective functioning of file recovery protection.

The TMF subsystem does not determine a schedule for online dumps. You must decide on an online dump schedule that satisfies the needs of your business operations. You can make online dumps without stopping your applications.

When scheduling online dumps, consider these guidelines:

- You can send online dumps to disk or to tape. Dumping to tape uses one tape drive completely and some system resources. You might not want to schedule online dumps and backups (described under Backup Strategies on page 4-17) at the same time or during the peak hours of application processing.
- You should coordinate online dumps with application activity. For example, if your site performs a series of batch processing or weekly updates at a particular time each week, you should follow those operations with online dumps of database

objects. Thus, if a file recovery is necessary your online dumps already reflect the batch updates. TMF would need to apply only those database changes that occurred after the online dumps were taken.

- When you create a new table and you want to provide file recovery protection for the table, you should make an initial online dump of the file after creating it.
- Certain DDL statements invalidate previous online dumps. For instance, whenever you load tables, upgrade catalogs, create new indexes, partition tables or indexes, or restructure or move the database, you should always make new online dumps to ensure the new status of the database is recorded correctly.

  For more information, see Operations That Invalidate TMF Online Dumps on page 11-15.
- Operations that use the WITH SHARED ACCESS option allow you to take online dumps while the operations are running.
- The TMF catalog can retain online dumps of several generations of each file. The number of generations retained depends on the RETAINDEPTH option of the TMF configuration parameters. Each generation of an online dump provides a starting point for a file recovery operation. You gain greater reliability by keeping extra generations of online dumps, but the site needs additional tape management for the online dumps and audit trails.

For additional information about TMF recovery operations, see the *TMF Operations and Recovery Guide*.

TMF Considerations in Using SQLCI

You can define and manage transactions from SQLCI, as follows:

- For DML statements, SQLCI generates TMF transactions for individual statements if the AUTOWORK session option is set to ON. If you set AUTOWORK to OFF, disabling automatic transaction generation for DML statements, you must explicitly begin and end TMF transactions in the SQLCI session. The SQL statements that explicitly control transactions are BEGIN WORK, COMMIT WORK, and ROLLBACK WORK. AUTOWORK ON is the default.
- With AUTOWORK set to either ON or OFF, you can explicitly define a TMF transaction, also called a user-defined TMF transaction. You can use a user-defined TMF transaction to ensure that several statements are either all executed successfully or all rolled back.

  These commands and statements make up a complete user-defined TMF transaction:

```
>>  BEGIN WORK;
>>  SELECT .....;
>>  INSERT .....;
>>  DELETE .....;
>>  COMMIT WORK;
```

- For DDL statements, the catalog manager generates the appropriate number of TMF transactions for the operations, reducing the overhead associated with TMF audit trails and ensuring that the necessary locks are acquired for the operations. These system-generated transactions occur regardless of the AUTOWORK setting. For DDL statements issued within a user-defined TMF transaction, the catalog manager does not initiate a system-defined TMF transaction.
- In addition to DDL statements, the COPY, PURGE, SECURE, and VERIFY commands run within a TMF transaction when they operate on audited objects.
- You cannot run a DUP, LOAD, or PURGEDATA command within a user-defined TMF transaction. You also cannot run a DDL statement or PURGE command within a user-defined TMF transaction or with a statement embedded within a program if the command operates on any nonaudited objects.
- You should not start a user-defined TMF transaction for the CREATE INDEX, CREATE CONSTRAINT, DOWNGRADE CATALOG, MOVE PARTITION, or UPDATE STATISTICS statement. For any of these operations, the catalog manager automatically starts several TMF transactions. Certain portions of the operation, however, are performed outside a TMF transaction unless you start one. If performed on one or more large tables within a TMF transaction, the operation could cause a TMF error.
- Only one user-defined TMF transaction can be active at a time in an SQLCI session. You must commit or roll back the current user-defined transaction before starting another.
- SQLCI provides an AUDITONLY option for the AUTOWORK ON command. If AUDITONLY is in effect, SQLCI releases locks only on audited tables and holds locks on nonaudited tables. You can use this option to hold locks on nonaudited objects throughout a series of transactions and then use the UNLOCK TABLE statement to release the locks on nonaudited tables. Using AUDITONLY helps reduce the possibility of a deadlock between audited and nonaudited table locking. AUTOWORK ON (without AUDITONLY) is the default.

Backup Strategies

Although you cannot determine whether you will ever need to use your backup tapes, you should schedule backups regularly as a general precaution. In addition, you should back up affected volumes or possibly the entire system when special events take place, such as equipment changes, configuration changes, and major software changes. You should also periodically back up to tape all nonaudited files, because no other method of recovery is available for these files.

This subsection contains general guidelines for backing up SQL objects stored as Guardian files, including information about these specific backup topics:

- Daily backups
- Periodic full backups
- Daily timestamp backups

- Backing up catalogs
- Backing up partitions
- Backing up indexes
- Backing up views
- Backing up collations
- Backing up by volume or by file
- Using OBEY command files for backup operations

For information about backing up OSS files, see the discussion of the pax utility in the *Open System Services Shell and Utilities Reference Manual*.

Note. The BACKUP and RESTORE recovery method differs from recovery methods provided by TMF. The BACKUP and RESTORE method is normally used as a secondary recovery scheme. Remember that an SQL object restored by the RESTORE utility might not be consistent with the current catalog description of the object.

When planning backup strategies, consider these guidelines:

- You should back up audited SQL objects by using TMF online dumps, as discussed under The TMF Subsystem on page 4-10. Preserving files by using BACKUP is not effective for recovery if any files are open during the BACKUP operation. Also, if you use the OPEN option, the image saved during the dump of the database might not be consistent. Before you begin a BACKUP operation, you should close and refresh the files by using PUP (D-series only) or SCF (G-series only). Files must remain closed throughout the BACKUP operation to ensure consistency.

- Although not recommended as the primary archiving method, you can use BACKUP to preserve audited in addition to the nonaudited files by using the AUDITED option. BACKUP and RESTORE can recover the database only to the time of the last backup; changes after that time are lost.

- You can back up entire tables and indexes or individual partitions. Also, you can specify indexes to be backed up or have them backed up automatically with the underlying tables. When you back up a table, index, or view, you must also back up any collations the object depends on because when the object is restored, it must use the same collations as before it was backed up.

- To archive nonaudited SQL tables, you must use BACKUP.

- SQL catalogs are not automatically dumped to tape in a BACKUP operation unless the SQLCATALOGS ON option is specified. Note, however, that the RESTORE utility cannot directly recover a catalog. All the catalog tables are audited so that they can be archived by using the TMF subsystem and recovered by using either TMF volume recovery or file recovery.

- Indexes backed up with INDEXES IMPLICIT, in effect by default, are not actually copied. The index definition is backed up; when restored, the index is re-created. Regardless of whether or not the index contained slack space, the restored index is re-created without slack space. To back up indexes and retain slack space, use the INDEXES EXPLICIT option.

- BACKUP has two formats for files: ARCHIVEFORMAT and DP2FORMAT. ARCHIVEFORMAT is the default used by BACKUP when a system includes SQL/MP. If you need to back up non-SQL files with DP2FORMAT, you must specify a file set list that does not contain any SQL objects and specify DP2FORMAT in the BACKUP command.
- If your system has only one tape drive, be careful not to perform a long backup at a time when the TMF subsystem might also need to dump an audit trail to tape. Before starting the backup, verify the status of the TMF audit trails to make sure no dump is currently queued. If the TMF subsystem reaches its maximum file limit during a period when the tape drive is unavailable for audit trail dumping, the TMF subsystem suspends transaction processing until an audit trail is dumped.
- If your system has multiple tape drives, you can use one drive for backups and another for the TMF audit-trail dumps.
- BACKUP accepts a qualified file set list for file-mode backups. You can use DEFINE names for the tape drive name and within the qualified file set list. BACKUP has many parameters that can improve the performance of tape handling, qualify the file set list, ignore errors, verify tape validity, and perform conversion between file types. For information about these parameters, see the *Guardian Disk and Tape Utilities Reference Manual.*

△ **Caution.** If an SQL object has the UNRECLAIMED FREESPACE (F) or INCOMPLETE SQLDDL OPERATION (D) attribute set, do not attempt to back up, move, or duplicate the object until the attribute is reset. For more information, see UNRECLAIMED FREESPACE (F) and INCOMPLETE SQLDDL OPERATION (D) Flags on page 7-24.

Daily Backups

To provide a high degree of protection, you can perform daily backups. Then the maximum amount of data lost from a failure never exceeds one working day.

A daily backup could be either a full backup of all files or a limited backup of specific files. If you use limited daily backups, you should also perform periodic full backups, as explained in Periodic Full Backups on page 4-20.

This example shows a BACKUP command to perform a full backup on all Enscribe files and on all SQL audited and nonaudited files (except catalogs) on the local node:

```
BACKUP $TAPE, *.*.* , AUDITED, OPEN, LISTALL
```

For recovering a volume separately, it can be helpful to perform the backup by volume name. This technique provides the same protection as the preceding command but separates each volume on a set of tape reels.

This example backs up volumes separately:

```
BACKUP $TAPE, $SYSTEM.*.*, AUDITED, OPEN, LISTALL
BACKUP $TAPE, $VOL1.*.*, AUDITED, OPEN, LISTALL
...
BACKUP $TAPE, $VOL9.*.*, AUDITED, OPEN, LISTALL
```

You might not need to back up subvolumes that the SIT SYSGEN tape can recover or the audit trails dumped to tape by TMF procedures. This example performs a full backup on all files except $SYSTEM.SYSTEM.*, $SYSTEM.SYSnn.*, and $AUDIT.TRAILS.*:

```
BACKUP $TAPE, *.*.* EXCLUDE ($SYSTEM.SYSTEM.*, $SYSTEM.SYSnn.*,
       $AUDIT.TRAILS.*), AUDITED, OPEN, LISTALL
```

For daily backups, you can also perform volume-mode backups, as explained under Using Volume-Mode or File-Mode Backup on page 4-24.

Periodic Full Backups

A full backup performed periodically might be adequate for protecting your database. The time between periodic backups should not exceed the maximum amount of work that would be acceptable to lose or redo if a catastrophic failure occurred.

This command performs a full backup on all files and includes the catalog tables:

```
BACKUP $TAPE, *.*.* , AUDITED, SQLCATALOGS ON, OPEN, LISTALL
```

Daily Timestamp Backups

For large databases, a full backup can be inefficient. For some applications, the amount of change to database files is uneven; some files might change frequently, while other files seldom change from day to day.

BACKUP provides a mechanism to perform a partial backup automatically on only those files that have changed since the last backup date. You can use the features of a qualified file set list to isolate certain objects, such as files with a specified user ID, files created or modified within a certain timestamp expression, or files with a certain file code. By using the WHERE clause of the qualified file set list, you can back up only SQL files that have been modified from a certain date. The PARTIAL parameter in the BACKUP command set is valid only for Enscribe files.

If you perform partial backups, you must perform a full backup periodically to ensure that all files have been saved.

This example uses a qualified file set list to restrict the backup to files that were modified since the date of the last full backup:

```
BACKUP $TAPE, (*.*.* WHERE MODTIME AFTER JAN 10 1989),
       LISTALL, OPEN, AUDITED
```

Using the FROM CATALOG Option for SQL Objects

The qualified file set list includes a parameter that can specify objects registered in specific catalogs. The FROM CATALOG option of a qualified file set list specifies that only SQL objects registered in the specified catalog are part of the file set list. No Enscribe files except SQL program files are processed by the FROM CATALOG option. The objects affected are programs, tables, indexes, views, collations, and partitions of tables and indexes; the catalog tables themselves are not backed up unless you also specify the SQLCATALOGS ON option.

By using the FROM CATALOG option for an SQL database, you can achieve a backup of a catalog or list of catalogs. This clause can be useful in maintaining backups of logical groupings of SQL objects as they are grouped in the catalogs.

You can use a DEFINE name for the catalog name.

This example uses a series of commands to perform a full backup on all files from each catalog specified in the FROM CATALOG clause, with the files from each catalog on a separate reel. In this example, audited objects would not be backed up because the AUDIT option is omitted.

```
BACKUP $TAPE, ( *.*.* FROM CATALOG $VOL1.CAT1), LISTALL, OPEN
BACKUP $TAPE, ( *.*.* FROM CATALOG $VOL2.CAT2), LISTALL, OPEN
...
BACKUP $TAPE, ( *.*.* FROM CATALOG $VOL9.CAT9), LISTALL, OPEN
```

This command performs a full backup on all files from several catalogs specified in the catalog list:

```
BACKUP $TAPE, (*.*.* FROM CATALOG ($VOL1.CAT1,$VOL2.CAT2, ...
            $VOL9.CAT9)), LISTALL, OPEN
```

Backing Up Partitions

You can use the PARTONLY ON option when a database has partitioned tables. PARTONLY enables you to back up and restore single components of a partitioned database.

PARTONLY could be applicable in these situations:

- A table partition resides on a remote node.
- A particular volume with partitioned objects is archived individually.
- A volume-mode backup is performed for archiving, and additional file-mode archiving is needed for individual partitioned files.

You cannot use the PARTONLY option with the MAP NAMES option.

△ **Caution.** Use extreme caution when using PARTONLY in BACKUP and RESTORE operations for partitioned files. It is possible to make the primary and secondary partitions of a file inconsistent both with each other and with indexes.

For example, if you delete a table partition after a BACKUP PARTONLY operation, a RESTORE PARTONLY operation would corrupt the table because the table no longer has that partition.

This example performs a full backup on each volume with the PARTONLY option:

```
BACKUP $TAPE, $SYSTEM.*.*, PARTONLY ON, OPEN, LISTALL
BACKUP $TAPE, $VOL1.*.*,   PARTONLY ON, OPEN, LISTALL
...
BACKUP $TAPE, $VOL9.*.*,   PARTONLY ON, OPEN, LISTALL
```

Backing Up Indexes

The INDEXES option controls whether indexes are backed up automatically when the underlying table is backed up. If you specify INDEXES IMPLICIT or use this option by default, index definitions are backed up automatically with the underlying table, regardless of whether it is explicitly named in the file set list. If you use the INDEXES EXPLICIT option, only those indexes named in the file set list are backed up, and they are copied in their entirety.

In either case, with either IMPLICIT or EXPLICIT, you must also back up any collations used by the indexes.

The INDEXES IMPLICIT option produces this error message from the BACKUP utility, whether you specify IMPLICIT or use it by default:

```
index-name  *ERROR* SQL index-tables handled implicitly
```

This error generates a count of the files not backed up at the end of the BACKUP operation; the count includes the indexes, because the index files are not actually copied. When the underlying table is restored, however, indexes are re-created from the catalog description, but the restored indexes are re-created without slack space, regardless of whether or not they contained slack space. The error occurs in the normal operation of BACKUP when INDEXES IMPLICIT is in effect.

The INDEXES EXPLICIT option could be applicable in these situations:

- You want to retain the slack space in an index when it is restored.
- The underlying table of a local index resides on a remote node.

Caution. Use extreme caution when applying INDEXES EXPLICIT in a BACKUP operation. Incorrect use of this strategy can result in making an index and its underlying table inconsistent with each other.
For example, if you backed up a table with the INDEXES EXPLICIT option, deleted the indexes, and then restored the table with the INDEXES EXPLICIT option, the table would be corrupt because the table's file label no longer has any information about the indexes.

Backing Up Views

Protection views are automatically backed up when you include the underlying table name in the file set list of the BACKUP command. Protection views cannot be backed up explicitly. In contrast, shorthand views are not automatically backed up unless you specify the view names in the file set list of the BACKUP command.

No other options of the BACKUP and RESTORE commands enable you to control the archiving of views.

When you back up a view, you must also back up any collations used by the view.

This example backs up the table EMPLOYEE, which has one protection view, EMPLIST, and two shorthand views, EMPSHV1 and EMPSHV2. The example shows the BACKUP command that backs up both protection and shorthand views:

```
BACKUP $TAPE,($VOL1.SVOL.EMPLOYEE,$VOL1.SVOL.EMPSHV1,
              $VOL2.SVOL.EMPSHV2), LISTALL, OPEN
```

Backing Up Collations

To back up collations, specify them in the file set list in the BACKUP command, as you do tables and indexes. Whenever you back up a table, index, or view that uses a collation, be sure to back up the collation as well, because when the object is restored, it must use the same collation.

For example, for a table that includes names from different countries, this command backs up a table, the indexes on a table, and the collations used by the table and the indexes:

```
BACKUP $TAPE, ($VOL1.INVENT.SUPPLIER, $VOL2.COLLS.COLLFR,
               $VOL3.COLLS.COLLJPN, $VOL4.COLLS.COLLSP,
               $VOL5.COLLS.COLLIND, $VOL6.COLLS.COLLKOR),
               INDEXES IMPLICIT, LISTALL, OPEN
```

If you have only a few collations used by many objects, you might want to make a one-time backup of all the collations and save the backup copies for restoring with dependent objects. For collations that do not change often, this one-time backup is easier to manage than repeated backups of the same collations with every dependent object.

Note. When restoring collations, they must be restored before their dependent objects.

Using Volume-Mode or File-Mode Backup

Choose a volume-mode or file-mode backup, depending on the results you want to achieve.

For a volume-mode backup, consider these characteristics of the operation:

- BACKUP makes a physical copy of a volume on tape. The copy includes the disk data structures that are usually not apparent to the user. When a volume-mode backup tape is restored, the entire disk volume, including the disk structures, is restored and duplicates the original state of the disk.
- Only the super ID can perform the operation.
- A volume-mode backup tape cannot be used to restore individual files to a disk.
- The operation copies all data files, SQL files, and SQL catalog tables. A volume-mode backup might be useful if you need to switch from one disk drive to another.
- Catalog consistency is assured only if the catalogs and the objects registered in each catalog are on the same volume.

If you plan to do a volume-mode backup, see the *Guardian Disk and Tape Utilities Reference Manual*.

For a file-mode backup, consider these characteristics of the operation:

- BACKUP copies each file defined in the file set list to tape. When a file is restored to a disk, the file is copied to the best logical free space. Restoring a file-mode backup tape to a clean disk compacts the files to use the free space on that disk most efficiently.
- BACKUP does not back up the SQL catalogs unless the BACKUP command includes the SQLCATALOGS ON option. RESTORE can restore the catalogs as SQL tables but not as catalog tables. To recover catalogs, you should use the TMF subsystem, as explained under Restoring Objects With TMF Recovery Operations on page 11-11.

Because SQL/MP can introduce dependencies between disk volumes, both file-mode and volume-mode backups can create inconsistent databases when RESTORE operations are applied without regard to these interdependencies.

Using OBEY Command Files for Recovery

SQL/MP maintains an active data dictionary and uses the TMF subsystem to protect the data dictionary by auditing the catalog tables. This dictionary, therefore, is subject to the same recovery issues as SQL tables.

You must protect the database in every way possible. Although the TMF subsystem protects the online database system, you might also want to use an offline method to increase your protection.

Another recovery method is to maintain EDIT files that contain database creation statements and commands for re-creating your database. When you first create the database, you should record the statements and commands in EDIT files. These EDIT files are called OBEY command files because you can use the OBEY command through SQLCI to run part or all the statements and commands in the files. OBEY command files provide both a simple method for entering the detailed database creation statements and commands and a method for backing up table, view, and index definitions.

Whenever you enter a DDL statement, the active data dictionary changes. You should always use the SQLCI logging facility when you make database changes, additions, or deletions. Keep the log file for your records. If you need to determine valid definitions, your original command file and a log of the changes can assist you in reconstructing data definitions.

You can use your recovery command files to re-create a database on the same system or on another system. You can also use these files as a verification point for the correctness of the catalog descriptions if the data dictionary changes inadvertently.

5 Creating a Database

The task of creating an SQL/MP database consists of creating catalogs, tables, views, indexes, collations, and constraints. Before you attempt these operations, however, you should understand planning schemes and define the database layout, as described in Section 3, Understanding and Planning Database Tables. You should have already planned your security, TMF requirements, and recovery mechanisms as described in Section 4, Planning Database Security and Recovery. Finally, you create the database itself.

After you create the database, you can load data into the base tables, compile your application programs, and perform database management operations.

This section includes security guidelines for object creation. The following user names and corresponding user IDs are used in the examples.

```
Super.Super             255,255
Super.Operator          255,001
DBA.Super               001,255
DBA.Dev                 001,100
DBA.Prod                001,200
Dev.MGR                 100,255
Dev.User                100,001
Prod.MGR                200,255
Appl.MGR                250,255
Appl.User               250,001
```

Creating Catalogs

An SQL/MP catalog consists of information stored in a set of tables that are indexed for quick access. When you create a catalog, you automatically create all its tables and indexes.

Each node on which SQL/MP is used must have at least one catalog. Each table, view, index, partition, collation, or catalog table located at a node must be described in a catalog on the same node.

Each volume on a node can have one or more catalogs. A given catalog on a volume can describe objects on any volume in the same node.

Each subvolume of a volume can have only one catalog. This catalog can describe objects on the same subvolume, on another subvolume of the same volume, or on another volume of the same node.

Each catalog has the same name as the subvolume on which it resides. Thus, if a catalog resides on subvolume SUBV1 of volume $VOL1 on node \SYS1, the full name of the catalog is \SYS1.$VOL1.SUBV1.

When you create a new catalog with the CREATE CATALOG statement, you must have authority to write to the CATALOGS table of the system catalog in which all catalogs on a system are registered.

A catalog has two types of components, described in these subsections:

- Catalog tables
- Indexes on catalog tables

Catalog Tables

Each SQL/MP catalog table describes either a particular type of object or some aspect of an object. For instance, the BASETABS catalog table describes base tables.

Each catalog table has a name assigned by the system; for example, BASETABS. Although the table names are the same in all catalogs, the full name of each catalog table is unique in the network. For example, the full name of the BASETABS catalog table in the SUBV1 catalog is \SYS1.$VOL1.SUBV1.BASETABS.

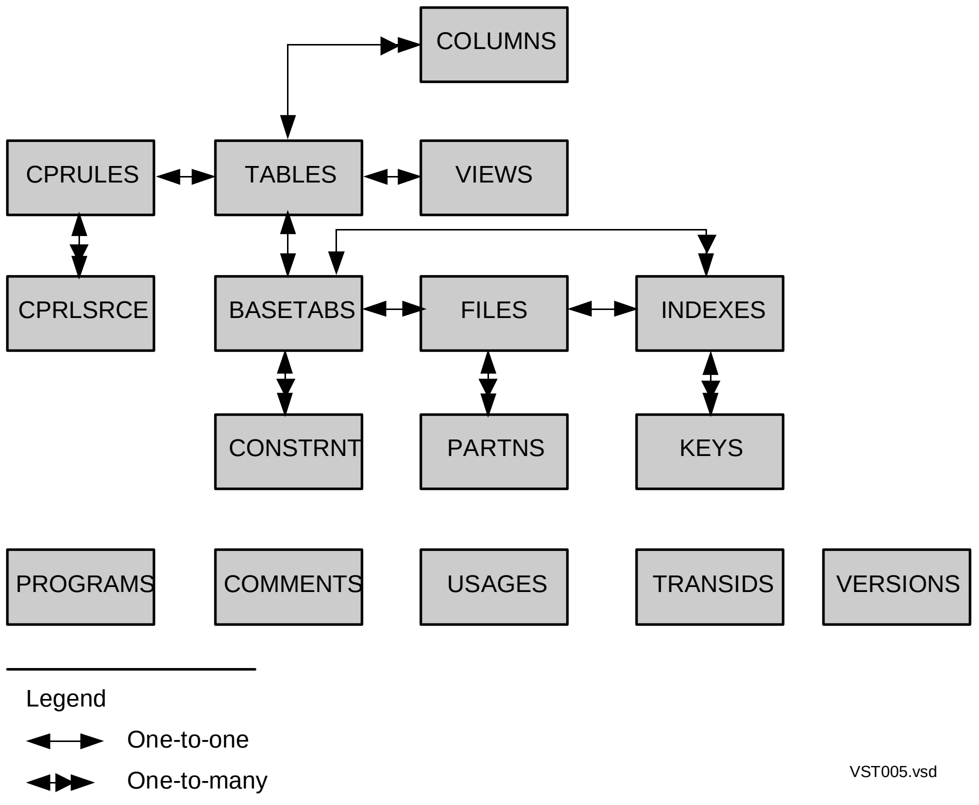
The individual catalog tables that make up a SQL/MP catalog are:

Table Name	**Table Function** (page 1 of 2)
BASETABS	Describes base tables (database tables but not views).
CATALOGS	Describes all catalogs on the system. This table is the system directory of catalogs and resides only in the system catalog.
COLUMNS	Describes the columns of each table listed in the TABLES catalog table.
COMMENTS	Contains comments on tables, views, indexes, constraints, and collations; comments on columns described in the catalog; and help text for columns.
CONSTRNT	Describes constraints defined on base tables.
CPRULES	Describes collations.
CPRLSRCE	Stores source text for character processing rules.
FILES	Describes attributes of files that contain tables and indexes.
INDEXES	Describes columns of primary keys and indexes.
KEYS	Describes the key columns of indexes.
PARTNS	Describes partitions of tables and indexes.
PROGRAMS	Describes object program files that have been SQL compiled.
TABLES	Describes tables and protection and shorthand views.
TRANSIDS	Keeps TMF transaction IDs for current DDL operations on the catalog. This information helps prevent multiple DDL operations from being executed on the same catalog at the same time within the same TMF transaction ID

Table Name	Table Function (page 2 of 2)
USAGES	Describes dependencies among SQL objects.
VERSIONS	Contains version information about the catalog. This is a backup copy of the version information in the CATALOGS table of the system catalog.
VIEWS	Describes views defined on base tables.

Figure 5-1 illustrates relations between catalog tables.

Figure 5-1. Catalog Structure



Indexes on Catalog Tables

The catalog tables that have indexes, their index names, and the names of indexed columns are listed in this table. These indexes are required by the system and are created automatically for the catalog:

Table Name	Index Name	Index Columns (page 1 of 2)
INDEXES	IXINDE01	INDEXNAME
PARTNS	IXPART01	PARTITIONNAME

Table Name	**Index Name**	**Index Columns** (page 2 of 2)
PROGRAMS	IXPROG01	GROUPID, USERID
TABLES	IXTABL01	GROUPID, USERID
USAGES	IXUSAG01	USINGOBJNAME, USINGOBJTYPE

Only IXINDE01 is a unique index.

The *SQL/MP Reference Manual* presents a detailed description of each catalog table.

Requirements for Catalogs

A system that includes SQL/MP can have one or more catalogs to describe an SQL database.

Following are the location-related requirements for SQL/MP catalogs on a system:

- All SQL objects on a system must be described in a catalog on the same system.
- Each system that includes SQL/MP must have at least one catalog.
- Each volume containing a catalog must be enabled and audited by the TMF subsystem, because catalogs are collections of audited SQL tables.
- Each audited volume on a system can have one or more catalogs or no catalogs. Each catalog on a volume can describe objects on that volume or on another volume on the same system.
- A single subvolume can have only one catalog. Each catalog has the same name as the subvolume that contains the catalog.

  For example, the full name of a catalog on the system \SYS1, volume $VOL1, and subvolume SUBVOL1 is \SYS1.$VOL1.SUBVOL1.
- Catalog tables cannot be partitioned.

Each system that uses SQL/MP has a catalog called the system catalog that contains information about all the catalogs on the system. The system catalog is like any other catalog, with these exceptions:

- The system catalog is established during the installation of SQL/MP. For more information, see Installing SQL/MP on page 2-2.
- The system catalog contains an additional table, CATALOGS, which is the system directory of catalogs.
- To obtain the name of a local or remote system catalog, use the GET CATALOG OF SYSTEM statement. (For more information, see the *SQL/MP Reference Manual* or SQLCI online help.)

Design Considerations

Because only a few rules apply to catalog location, many possible schemes exist for creating catalogs. Before deciding how to create your catalog structure, consider these performance issues for catalogs:

- Dependent relationships in the USAGES tables of catalogs should be easily accessible. Dispersing tables, views, and indexes among catalogs does not improve throughput.
- A small performance penalty is associated with the use of multiple catalogs: a one-time penalty per process at table-open time for each catalog accessed for SQL compilation. This penalty applies for both DDL and DML statements. Because this penalty affects both explicit SQL compilation and automatic recompilation at run time, the penalty might be significant.
- The number of catalogs does not affect the collective number of rows stored in the catalog or catalogs, with one exception: When an object described in a catalog is related to an object described in a different catalog, the USAGES relationship is stored in both catalogs.

Multiple-Catalog Approach

A multiple-catalog approach, in which all objects are registered in the system catalog, has these advantages:

- The multiple-catalog approach simplifies security considerations if different user groups or applications have different security requirements.
- Concurrency problems are reduced if many operations use one catalog.
- If the catalog becomes unavailable because of a down or damaged volume, SQL compilations that require the catalog are suspended until the catalog is available.

  When SQL catalogs are unavailable, users cannot enter any DDL statement, dynamic SQL statement, or SQLCI command that requires the catalog or that starts an explicit SQL compilation. Also, programs cannot be automatically recompiled; only explicitly compiled programs can run, if they do not require any recompilations. The use of multiple catalogs reduces the effect of a single catalog becoming unavailable.

Single-Catalog Approach

The advantages of a single-catalog approach are:

- When using a single catalog, you do not need to unite the information in multiple catalogs to generate a single report on catalog information. (You can, however, join multiple catalogs by using the UNION operator to obtain a unified report across catalogs. Because the catalog tables are identical for all catalogs, these tables are ideal for use with the UNION operator.)

- Administrative control is easier, and perhaps safer, with fewer catalogs. If restricting the authority for users to create SQL objects is important, controlling fewer security strings is easier.

If you are using the $SYSTEM default location for the system catalog, you might want to limit use of the catalog to minimize SQL/MP disk accesses on $SYSTEM during SQL compilations. In this situation, you would not use the system catalog as a general-purpose catalog for applications, but only as a catalog directory, and you would have one or more other catalogs for your applications.

Note. The recommendation for the number of catalogs is to create the smallest number of catalogs you need for your business operations. Typically, you should establish catalog boundaries along the lines of application and user security requirements. Associate catalogs with sets of tables that are logically associated or that are used together.

Performance Considerations

The disk process cache setting for a disk that contains an SQL catalog affects the performance of SQL DDL statements. The system administrator should set the disk process cache to an appropriate value by using the PUP SETCACHE command (D-series only) and the SCF ALTER DISK, CACHE command (G-series only). This action is especially useful for catalogs that store information about tables with many partitions. The performance of DDL statements such as CREATE TABLE, ALTER TABLE ADD PARTITION, and DROP TABLE can be greatly enhanced with an effective cache setting.

For example, a table with 200 partitions, all described in a single catalog, has 40,000 rows in the PARTNS catalog table and in the IXPART01 index on the PARTNS catalog table. Creating such a table causes more than 80,000 writes to the catalog. Using the default cache value can cause this operation to take up to 25 times longer than if you set disk cache to 4 MB.

For more information about cache memory, see Managing Cache Memory Size on page 14-17. For information about PUP, see the *Peripheral Utility Program (PUP) Reference Manual* (D-series only). For more information about SCF, see the *SCF Reference Manual for G-Series RVUs*.

Creating a Catalog

This example creates a catalog on $VOL1.SALES, the current default volume and subvolume:

```
>>  CREATE CATALOG;
--- SQL operation complete.
```

You can use the SQLCI ENV command to list information about the current environment.

When creating catalogs in SQLCI, you must be aware of the SQLCI session environment. When you have not specified a current catalog in a session and you do not explicitly specify a catalog in the CREATE CATALOG statement, SQL/MP creates the catalog on the current default volume and subvolume.

This example specifies the location of the catalog. The catalog name is the subvolume name.

```
>>  CREATE CATALOG \SYS1.$VOL1.MFG;
--- SQL operation complete.
```

You can use a DEFINE name to specify a catalog name. In this example, the catalog is created on \SYS1.$VOL1.MFG as defined by the DEFINE =MFG. The INFO DEFINE command displays the DEFINE, showing the actual catalog name.

```
>>  INFO DEFINE =MFG;

      DEFINE NAME             =MFG
      CLASS                   CATALOG
      SUBVOL                  \SYS1.$VOL1.MFG

>>  CREATE CATALOG =MFG;
--- SQL operation complete.
```

If you are running SQL/MP on a system using the SMF product and you want to ensure that you can fall back to a non-SMF system, make sure that a given catalog's tables reside on one physical volume. If you specify a virtual volume for a catalog, SMF can distribute the catalog tables among multiple physical volumes in the storage pool. When this configuration is in place, there is no guarantee that you can return to using a nonvirtual volume. When you are certain you will not need to fall back to a non-SMF system, you can specify a virtual volume for a catalog without being concerned with the physical location of the files.

To ensure that a catalog's tables reside on one physical volume, you can specify a direct volume that is not in any storage pool, or you can use the PHYSVOL option, as follows:

```
>> CREATE CATALOG $virtual_vol.subvol PHYSVOL $physical_vol;
```

With the PHYSVOL option, you specify only the volume name. Also, the virtual volume specified with the CREATE CATALOG clause must be associated with the same storage pool that contains the physical volume specified with PHYSVOL. For more information about using this option, see the *SQL/MP Reference Manual* and the *Storage Management Foundation User's Guide*.

Securing Catalog Tables

When you create a catalog, SQL/MP assigns the catalog ownership to your Guardian user ID with your default security, unless you specify the SECURE attribute in the CREATE CATALOG statement.

The catalog tables compose the data dictionary, a vital part of an application's integrity. The security of a catalog should protect the data dictionary information from unauthorized removal or alteration.

If you specify a security string in the CREATE CATALOG statement, you must specify the catalog name. This example specifies the location of a new catalog and the security:

```
>>  CREATE CATALOG \SYS1.$VOL1.MFGCAT SECURE "GGNO";
--- SQL operation complete.
```

When you create a catalog or alter the security string for a catalog, the catalog security applies to all the catalog tables. If you do not specify a security string in the CREATE CATALOG statement, SQL/MP assigns your current default security to the catalog and all the catalog tables.

Access to Catalog Objects

Allowing access to the catalog does not automatically allow access to the objects described in that catalog. Access to the catalog is required in addition to access to the objects for DDL statements, DML statement compilations for SQLCI and dynamic SQL, most utility commands, and SQL program compilations.

Altering Security

To make the USAGES, TRANSIDS, and PROGRAMS tables accessible for SQL compilations of programs, you might need to change the security of each table in an ALTER TABLE statement. During explicit SQL compilation, any dependencies that a program has on tables or views described in a catalog are recorded in the catalog’s USAGES table. To insert the dependency record into the USAGES table, the catalog manager must start a TMF transaction, which is registered in the TRANSIDS table. Write access to the PROGRAMS table is required so that the SQL compiler can register programs in the table.

You can change the catalog security at creation time by specifying the SECURE attribute in the CREATE CATALOG statement. You can also change the security of these individual tables at any later time by using the ALTER CATALOG statement:

- CATALOGS (system catalog only)
- USAGES
- TRANSIDS
- PROGRAMS

If you use the SECURE attribute, you must specify a security string that gives the owner of the catalog tables read access.

For a user to compile a program, the user needs read and write access to the USAGES and TRANSIDS tables in any catalog containing descriptions of tables, views, collations, partitions, and indexes that the program uses in addition to write access to the PROGRAMS table of the catalog in which the program is registered.

Examples

These examples show access to the catalog tables. Actual access for certain statements can depend on the security of a table, view, or index.

The first example shows security that enables any network user to read or write to the catalogs in which objects are registered. Any network user can compile and register programs in this catalog and can create tables, views, and indexes. Only the super ID user in the DBA user group can drop the catalog.

```
$VOL1.SUBVOL catalog         Owner    = 001,255
                             Security = "NNNO"
```

The next example shows security that enables any group 100 user to compile programs that use tables and views described in the $VOL1.APPLPGM or $VOL2.APPLCAT catalog:

```
$VOL1.APPLPGM catalog        Owner    = 100,255
                             Security = "GGNO"
$VOL2.APPLCAT catalog        Owner    = 001,255
                             Security = "NGOO"
$VOL2.APPLCAT.USAGES         Owner    = 001,255
                             Security = "NNOO"
$VOL2.APPLCAT.TRANSIDS       Owner    = 001,255
                             Security = "NNOO"
```

All programs for this application are registered in the catalog $VOL1.APPLPGM. Any group 001 user can:

- Read or write to the catalog $VOL2.APPLCAT in which the objects are described
- Create tables, views, and indexes registered in this catalog (local users only)
- Execute programs registered in $VOL1.APPLPGM

Any network user can query descriptions in the catalog $VOL2.APPLCAT. Only the super ID user of each group can drop the catalog.

The next example shows security that enables any network group 001 user to read or write to the catalog in which the objects are described or to execute programs registered in the catalog $VOL2.APPLCAT. Only local group 001 users can create dependencies on any objects described in the catalog or compile programs that use any tables or views described in the catalog.

```
$VOL2.APPLCAT catalog        Owner    = 001,255
                             Security = "CCCO"
$VOL2.APPLCAT.USAGES         Owner    = 001,255
                             Security = "CGOO"

$VOL2.APPLCAT.PROGRAMS       Owner    = 001,255
                             Security = "CGOO"
$VOL2.APPLCAT.TRANSIDS       Owner    = 001,255
                             Security = "CGOO"
```

Securing the System Catalog

The system catalog maintains the directory of catalogs on each system in the CATALOGS table. Except for this table, the system catalog is like any other catalog on the system. You can use the system catalog as a catalog directory only or as a general-purpose catalog.

The security of the system catalog should protect the catalog from removal.

CATALOGS Table

For a user to create other catalogs on the system, the user must have authority to write to the system directory of catalogs, the SQL.CATALOGS table. You can secure this table separately from the rest of the system catalog to restrict the capability to create catalogs within your application.

You might consider giving read authority to all users, enabling them to query the SQL.CATALOGS table.

Examples

This example shows catalog security that ensures that catalogs can be created only by the local database administrator's group (DBA.Super, DBA.Dev, and DBA.Prod):

```
System catalog ($SYSTEM.SQL)  Owner    = 001,255
                              Security = "OOOO"
$SYSTEM.SQL.CATALOGS          Owner    = 001,255
                              Security = "GGOO"
```

This example shows catalog security that gives any network user remote read access. Any user in the database administrator's user group can create catalogs on this system, either locally or remotely.

```
System catalog ($SYSTEM.SQL)  Owner    = 001,255
                              Security = "OOOO"
$SYSTEM.SQL.CATALOGS          Owner    = 001,255
                              Security = "NCOO"
```

Creating Base Tables

Base tables are the foundation of an SQL/MP database. All data physically resides in the base tables. When you create a table with the CREATE TABLE statement, you specify the definition of each data column and the attributes of the physical file in which the table is to be stored. Carefully consider the file attributes to ensure that the table will meet the needs of your application.

The CREATE TABLE statement stores the table definition in the specified SQL catalog and creates the table, which physically exists as a disk file. Before creating a table, you should understand the three types of table organizations and column, key, and index design considerations. For more information, see Understanding SQL File Structures on page 3-1.

For information about loading base tables, see Guidelines for Loading Tables on page 8-8.

Determining the Organization of the Physical File

When you create a table, you can use the ORGANIZATION clause in the CREATE TABLE statement to organize the physical file. The ORGANIZATION clause is optional.

The file organization can be key-sequenced, entry-sequenced, or relative. If you do not choose a file organization by using the ORGANIZATION clause, the organization defaults to key-sequenced.

Creating Key-Sequenced Tables

When you define a key-sequenced table, you also define the primary key used to access rows in the table. The data type, physical ordering, and primary key type play a role in the primary access sequence of the table.

Defining Primary Keys

Use these commands to define primary keys:

- User-defined primary key: specify columns of the primary key in the PRIMARY KEY clause of the CREATE TABLE statement. A user-defined primary key can include a number of contiguous or noncontiguous columns but cannot exceed 255 bytes.
- System-defined primary key: this is the default type of key generated by SQL/MP if you do not specify a PRIMARY KEY or CLUSTERING KEY clause.
- Clustering key: specify columns of the primary key in the CLUSTERING KEY clause of the CREATE TABLE statement. To this group of columns, SQL/MP appends a SYSKEY column to form a unique primary key.

Creating a Key-Sequenced Table With a User-Defined Primary Key

This example creates a key-sequenced table with a user-defined primary key on the current default subvolume and registers the table in the current default catalog. You can also use the ENV command to list the current environment before using the CREATE TABLE command.

```
>>  CREATE TABLE ORDERS
+>    (ORDERNUM    DECIMAL (6) UNSIGNED  NO DEFAULT NOT NULL,
+>     ORDER_DATE  DATETIME YEAR TO DAY  NO DEFAULT NOT NULL,
+>     DELIV_DATE  DATETIME YEAR TO DAY  NO DEFAULT NOT NULL,
+>     SALESREP    DECIMAL (4) UNSIGNED  DEFAULT SYSTEM,
+>     CUSTNUM     DECIMAL (4) UNSIGNED  NO DEFAULT NOT NULL,
+>      PRIMARY KEY ORDERNUM)
+>     EXTENT (100,100)
+>     BLOCKSIZE  4096
+>     MAXEXTENTS 24

+>     ORGANIZATION KEY SEQUENCED
+>     SECURE "GGOO";
--- SQL operation complete.
```

Creating a Key-Sequenced Table With a Clustering Key

If you want the rows in a key-sequenced table ordered by a column or combination of columns whose values do not uniquely identify rows, you can specify these columns as a clustering key in the CLUSTERING KEY clause of the CREATE TABLE statement. A clustering key is part of the primary key; SQL/MP adds a system-defined SYSKEY column to the clustering key to make the primary-key value in each row unique. The total key length, including the SYSKEY column, cannot exceed 255 bytes.

This example shows a table created with a CLUSTERING KEY definition that consists of the ORDERITEM and ORDERNUM columns. Internally, the actual primary key used will consist of the ORDERITEM, ORDERNUM, and SYSKEY columns.

```
>>  CREATE TABLE ODETAIL
+>    (ORDERITEM   DECIMAL (6) UNSIGNED  NO DEFAULT NOT NULL,
+>     ORDERNUM    NUMERIC (6) UNSIGNED  NO DEFAULT NOT NULL,
+>     ORDER_DATE  DATETIME YEAR TO DAY  NO DEFAULT NOT NULL,
+>     DELIV_DATE  DATETIME YEAR TO DAY  NO DEFAULT NOT NULL,
+>     SALESREP    DECIMAL (4) UNSIGNED  DEFAULT SYSTEM,
+>     CUSTNUM     DECIMAL (4) UNSIGNED  NO DEFAULT NOT NULL)
+>      CLUSTERING KEY (ORDERITEM,ORDERNUM)
+>     EXTENT (100,100)
+>     BLOCKSIZE  4096
+>     MAXEXTENTS 64
+>     ORGANIZATION KEY SEQUENCED
+>     SECURE "GGOO";
--- SQL operation complete.
```

If you do not specify the organization in the CREATE TABLE statement, the organization defaults to key sequenced.

Creating a Key-Sequenced Table With Dependent Objects

Example 5-1 on page 5-13 creates a table and a set of objects that depends on the table. The example uses DEFINE names in SQL statements. The INFO DEFINE command displays the DEFINEs. You would usually enter this set of commands into an EDIT file you could use as an OBEY command file within SQLCI.

Example 5-1. Creating a Table and Dependent Objects (page 1 of 2)

```
--- DEFINEs were previously added during this SQLCI
--- session or inherited from the command interpreter.
>> INFO DEFINE =MCAT;
     DEFINE NAME             =MCAT
     CLASS                   CATALOG
     SUBVOL                  \SYS1.$VOL1.MFG
>> INFO DEFINE =ORDERS;
     DEFINE NAME             =ORDERS
     CLASS                   MAP
     FILE                    $VOL1.MFG.ORDERS
>> INFO DEFINE =REPORDS;
     DEFINE NAME             =REPORDS
     CLASS                   MAP
     FILE                    $VOL1.MFG.REPORDS

>> INFO DEFINE =XORDCUS;
     DEFINE NAME             =XORDCUS
     CLASS                   MAP
     FILE                    $VOL2.MFG.XORDCUS
     BLOCKSIZE  4096
>> CREATE TABLE =ORDERS
+>    (ORDERNUM    DECIMAL (6) UNSIGNED  NO DEFAULT NOT NULL,
+>     ORDER_DATE  DATETIME YEAR TO DAY  NO DEFAULT NOT NULL,
+>     DELIV_DATE  DATETIME YEAR TO DAY  NO DEFAULT NOT NULL,
+>     SALESREP    DECIMAL (4) UNSIGNED  DEFAULT SYSTEM,
+>     CUSTNUM     DECIMAL (4) UNSIGNED  NO DEFAULT NOT NULL,
+>      PRIMARY KEY ORDERNUM)
+>  EXTENT (100,100)
+>  MAXEXTENTS 24
+>  CATALOG =MCAT
+>  SECURE “GGOO”;
--- SQL operation complete.
>> CREATE CONSTRAINT DATE_CONSTRNT
+>     ON =ORDERS
+>     CHECK DELIV_DATE >= ORDER_DATE;
--- SQL operation complete.
>> COMMENT ON TABLE =ORDERS
+>    IS “ACTIVE ORDERS TABLE”;
--- SQL operation complete.
>> COMMENT ON COLUMN ORDER_DATE
+>    ON =ORDERS
+>    IS “FORMAT IS YYMMDD”;
--- SQL operation complete.
>> COMMENT ON COLUMN DELIV_DATE
+>    ON =ORDERS
+>    IS “FORMAT IS YYMMDD”;
--- SQL operation complete.
```

Example 5-1. Creating a Table and Dependent Objects (page 2 of 2)

```
>> CREATE VIEW =REPORDS
+>    AS SELECT SALESREP,ORDERNUM,DELIV_DATE,ORDER_DATE
+>    FROM =ORDERS
+>    CATALOG =MCAT
+>    FOR PROTECTION;
--- SQL operation complete.
>> CREATE INDEX =XORDCUS
+>    ON =ORDERS (CUSTNUM)
+>    KEYTAG “OC”
+>    EXTENT (100,50)
+>    BLOCKSIZE  2048
+>    MAXEXTENTS 24
+>    ICOMPRESS
+>    CATALOG =MCAT;
--- SQL operation complete.
```

Creating Entry-Sequenced Tables

Entry-sequenced files are designed for sequential access. They consist of variable-length records. New records are always appended to the end of the file; as a result, the records in the file are arranged physically in the order in which they were added to the file. Existing records can be updated, but they cannot be deleted. A user performing update operations can update rows but cannot delete them and cannot lengthen or shorten values in varying-length columns (VARCHAR, NCHAR VARYING).

This example creates an entry-sequenced table. This table is nonaudited, and the primary extent is preallocated to ensure enough space.

```
>>  CREATE TABLE \SYS1.$VOL3.LOGS.TRANSEQ
+>     (TRANSEQ_NUM       NUMERIC (10)  NO DEFAULT NOT NULL,
+>      CLASS_CODE        PIC 9(4)      DEFAULT SYSTEM,
+>      CLASS_STATUS      NUMERIC (6)   DEFAULT SYSTEM,
+>      STATUS_MSG        PIC X(45)     DEFAULT SYSTEM)
+>     EXTENT (10000,1000)
+>     CATALOG \SYS1.$VOL3.ADMIN
+>     ALLOCATE 1
+>     NO AUDIT
+>     ORGANIZATION ENTRY SEQUENCED;
--- SQL operation complete.
```

Creating Relative Tables

Relative files consist of fixed-length physical records accessed by relative record number. A record number is an ordinal value and corresponds directly to the record’s position in the file. The first record is identified by record number zero. Succeeding records are identified by ascending record numbers in increments of one.

You can refer to specific records within a relative table either by their primary key (relative record number) or by the content of other key fields denoted by an index, such as a department number or zip code in an employee table. This reference is made through the WHERE clause in a DELETE, SELECT, or UPDATE statement.

A user performing update operations can update or delete rows and can lengthen or shorten values in varying-length columns. If the logical length of the record varies, however, the physical space consumed in a relative table is always the same. Moreover, all blocks allocated for a relative table are always full, even if the table includes zero-length records.

This example creates a relative table. RECLENGTH is specified as 100 to allow space for adding columns later.

```
>>  CREATE TABLE \SYS1.$VOL3.CODS.HCODES
+>     (CODENUM           PIC 9(4)    NO DEFAULT NOT NULL,
+>      ORGANIZATION      PIC X(10)   DEFAULT SYSTEM,
+>      USAGES_CODE       NUMERIC (6) DEFAULT SYSTEM,
+>      DESCRIPTION       PIC X(45)   NO DEFAULT NOT NULL)
+>     EXTENT (10,10)
+>     BLOCKSIZE  2048
+>     RECLENGTH  100
+>     AUDIT
+>     CATALOG \SYS1.$VOL3.ADMIN
+>     ORGANIZATION RELATIVE;
--- SQL operation complete.
```

Determining the Number of Records per Block

The BLOCKSIZE attribute in the CREATE TABLE statement lets you specify the block size for SQL tables. The maximum block size of 4096 is recommended. Choice of block size can affect the performance of your database. For additional performance information, see Specifying Block Sizes for Files on page 14-24.

Key-Sequenced Tables

The maximum record size for a key-sequenced table is the block size less 32 bytes for block header information. In addition, each record in a block requires two bytes to store the record’s offset location from the block header. Thus, for the maximum block size of 4096, the maximum usable record size is 4062 bytes if you store one record per block.

To determine the number of records that can be guaranteed to fit in each block, use this formula, in which N is the number of records, B is the block size, and R is the record length:

```
N = (B - 32) / (R + 2)
```

Thus, if your record length is 202 bytes and the block size is 4096, you can compute the number of records per block as follows:

```
N = (4096 - 32) / (202 + 2) = 19
```

For a key-sequenced table, the number of bytes allocated for a row equals the number of bytes in the row when the row is inserted into the file. After a row has been inserted, its length can be changed by updates that change values of varying-length columns.

Thus, the actual number of rows stored in a block might be greater than your calculated value of N if VARCHAR columns use fewer bytes than their maximum byte length.

Although the size of a VARCHAR column can change, the total row length cannot exceed the specified maximum record size for the table.

Moreover, if applications will update and insert records, you will want to leave free space in the block to avoid too-frequent block splits, which eventually fragment the file, use unnecessary storage space, and, possibly, affect performance. The SLACK, ISLACK, and DSLACK options in the SQLCI LOAD command and in the online FUP RELOAD command allow you to specify the amount of free space that will be left in each block when records are loaded into the table. For more information about these options, see Reorganizing a Database Online on page 8-2 and Loading, Copying, Appending, and Purging Data on page 8-7.

Entry-Sequenced and Relative Tables

The maximum record size for an entry-sequenced or relative table is the block size less 22 bytes for block header information. In addition, each record in a block requires two bytes to store the record’s offset location from the block header. Thus, for the maximum block size of 4096, the maximum usable record size is 4072 bytes if you store one record per block.

To determine the maximum number of records that will fit in each block (for an entry-sequenced table) or fill each block (for a relative table), use this formula, in which N is the number of records, B is the block size, and R is the record length:

```
N = (B - 22) / (R + 2)
```

If your record length is 35 bytes and the block size is 4096, you can compute the number of records per block as follows:

```
N = (4096 - 22) / (35 + 2) = 110
```

Additional Guidelines for Creating Tables

Consider these additional guidelines when creating tables:

- Specify PARTITION ARRAY EXTENDED to take advantage of the greater number of partitions and indexes available for tables and indexes on versions 320 and later of SQL/MP software. Note, however, that DML and DDL statements on tables and indexes with extended partition arrays can only be performed from nodes running version 320 or later of SQL/MP software.
- Specify table attributes that are best for the performance, access, size, and protection of the data in the base table:
    - Use BLOCKSIZE, EXTENTS, MAXEXTENTS, ALLOCATE, ICOMPRESS, DCOMPRESS, and RECLENGTH, if applicable, for controlling the size of the table.

  - Use AUDIT to protect the table with TMF auditing. If neither AUDIT nor NO AUDIT is specified, AUDIT is assigned by default.
  - Use AUDITCOMPRESS to minimize the amount of audit-trail resources required. Use NO AUDITCOMPRESS if you need to read the complete before-images and after-images directly from the audit trails.
  - Use BUFFERED, SERIALWRITES, and VERIFIEDWRITES to control the disk processing of the table.
  - Use CLEARONPURGE, SECURE, and NOPURGEUNTIL to control the security and the ability to write to or purge (drop) a table.

- Create tables from EDIT files that you use as OBEY command files within SQLCI. These EDIT files store the data definitions outside the data dictionary and make the definitions available for repeatable operations, if necessary. Because CREATE TABLE statements can be very long, it is easier to correct errors in an EDIT file than interactively in SQLCI.
- Use class MAP DEFINE names to identify the actual table names. The use of DEFINEs allows mobility of the EDIT command files: you can use the same files to create tables on different volumes and systems. For more information about DEFINEs, see Using DEFINEs on page 10-30.
- Consider creating dependent objects at the same time you create a table. To simplify these operations, you can put all the statements (such as CREATE TABLE, CREATE VIEW, CREATE CONSTRAINT, COMMENT, and CREATE INDEX) in the same EDIT file.
- To specify the catalog in which the table is to be registered, include the CATALOG option in the CREATE TABLE statement if this catalog is different from the default catalog. You can use a class CATALOG DEFINE name to identify the target catalog. The target catalog must be an existing catalog.
- To create a table exactly like an existing table, use the LIKE option in the CREATE TABLE statement. The new table always contains the same column structure as the source table. The new table is not partitioned, however, even if the source table is partitioned, unless you use the PARTITION clause when creating the table. In such a case, the target table has partitions even if the source table does not.

  Optionally, you can create the new table with the same comments, constraints, headings, and help text as the source table. Alternatively, you can override these attributes and create the new table with different comments, constraints, headings, and help text.

  The new table inherits collation information from the existing table. The columns in the new table cannot refer to different collations.
- If you plan to use similarity checking with the table, be sure to use the CREATE TABLE statement's SIMILARITY CHECK ENABLE clause.

Creating Tables on a System That Uses SMF

If you are running SQL/MP on a system using the SMF product, you can specify a virtual volume for a table. The virtual volume is associated with a storage pool; SMF places the table on a physical volume in that storage pool. SMF chooses a physical location based on its size estimate of the file and on the available space in the pool.

In exceptional cases, you might want a file to reside on a particular physical volume. To accomplish this, you can specify a direct volume that is not in any storage pool, or you can specify a virtual volume for the table and use the PHYSVOL option to select a particular physical volume in the pool.

If you do need to specify a physical volume for a file, it can be advantageous to use the PHYSVOL option rather than specifying a direct volume outside the control of SMF. If you use the PHYSVOL option, you can move the file to a different physical location in the future without having to recompile query execution plans or change other SQL objects (such as indexes, views, and catalog tables) that refer to the file. You can move the file even if a referencing object—an index, for example—is unavailable.

However, if the file is on a direct volume and you move the file with the ALTER TABLE. . . MOVE statement, SQL changes the external references to the file. In this situation, the referencing objects must be available.

This example creates a table on a virtual volume, $VIR1, associated with a SMF pool. The example specifies that the file reside on the physical volume, $PVOL3:

```
>>  CREATE TABLE $VIR1.MFG.ORDERS
+>    (ORDERNUM    DECIMAL (6) UNSIGNED  NO DEFAULT NOT NULL,
+>     |
+>     PRIMARY KEY ORDERNUM)
+>     PHYSVOL $PVOL3
+>     |
+>     ORGANIZATION KEY SEQUENCED;
--- SQL operation complete.
```

In the preceding example, $VIR1 must be associated with the storage pool that contains $PVOL3. When you create a table name on a virtual volume such as $VIR1, you use the same syntax you would use for volumes not managed by SMF. However, with the PHYSVOL option, you specify only the volume name (such as $PVOL3). SQL returns an error if you specify a full *`volume.subvolume.file`* name with the PHYSVOL option. If you omit the PHYSVOL option, SMF determines the physical volume on which the ORDERS table resides.

For more information about using this feature, see the *SQL/MP Reference Manual* and the *Storage Management Foundation User’s Guide*.

Defining Columns

To ensure the validity of your database, you must first define columns correctly for the use of the data and assign data types that provide the best design for your application. It is the database administrator’s task to consider how the data is used and to assign appropriate data types and constraints.

To define columns for a table, specify the column definitions in the CREATE TABLE statement or in ALTER TABLE statements with the ADD COLUMN clause. When you define a column, you specify the column name, data type, and, optionally, other column attributes.

For information about constraints, see Creating Constraints on Data on page 5-51.

Specifying Column Names

When naming columns, consider these guidelines:

- A column name is an SQL identifier that can contain at most 30 of these characters: letters (A-Z, a-z), digits (0-9), and the underscore (_). The name must begin with a letter. SQL/MP reserved words, listed in the *SQL/MP Reference Manual*, are not allowed as column names.
- Column names should be descriptive names for your application to help programmers and users remember the names correctly.

  These examples of column definitions show the column names, column descriptions, and default values:

```
LOCATION            PIC X(20)         DEFAULT SYSTEM   NOT NULL
AREA                CHAR (3)          DEFAULT SYSTEM   NOT NULL
STATE               CHAR (2)          DEFAULT "MO"     NOT NULL
PHONE               PIC 9(7)          DEFAULT SYSTEM   NOT NULL
LAST_AMOUNT         PIC S9(6)V99 COMP DEFAULT SYSTEM   NOT NULL
CHANGE_DATE         DATETIME          YEAR TO MINUTE   NOT NULL
EMPLOYEE_NUMBER     NUMERIC (4)       NO DEFAULT       NOT NULL
JOB_CODE_NUMBER     SMALLINT          DEFAULT SYSTEM   NOT NULL
TYPEJOBINCODE       NUMERIC (4)       UNSIGNED   NO DEFAULT
TYPEJOBDESCRIPTION  VARCHAR(20)       NOT NULL
LAST_VALUE          INT    UNSIGNED   DEFAULT SYSTEM   NOT NULL
```

- Column names can be specified in the CREATE TABLE statement in any combination of uppercase and lowercase letters. For example, these three column names are equivalent: LOCATION, Location, and location.

Specifying Data Types for Columns

Three basic formats of data can be stored in columns:

- Character and numeric data
- Binary numeric data
- Date, time, and time interval data

When determining the data type and attributes for a column, consider these guidelines:

- SQL/MP supports the ASCII character set and several other character sets for character data. For more information, see Defining Character Data on page 5-21.
- Specify a column default value for each column. This default value must be DEFAULT, DEFAULT SYSTEM, NOT NULL, NULL, NO DEFAULT, DEFAULT NULL, or LITERAL `literal`, or a valid combination of these values. For more information, see Using Default and Null Values on page 5-26.
- Specify any of these attributes for your application's use, if applicable: HEADING, HELP TEXT, and UPSHIFT. For more information, see Specifying Column Attributes on page 5-28.
- Collation of single-byte character data is performed in the order represented by the ordinal positions of the characters in the ASCII set. Alternatively, you can specify a different collating sequence for single-byte character data by creating a collation object and associating the collation with the character data. If you do associate a collation with the column, the character set associated with the collation must be the same as the character set defined for the column.
- Collation of numeric values occurs with negative numbers preceding positive numbers.
- For sorting, the null value is considered to be greater than all other values.
- For compatibility of SQL/MP data types, any character string type can be compared with all other character data types, and any numeric data type can be compared with all other numeric data types in DML comparison expressions. Character strings and numeric data types, however, are not compatible with each other; they cannot be compared directly by SQL/MP during a retrieval using predicates.

  Comparisons between character strings and numeric data types can occur only within user-written application code or, for parameters, by using the CAST function. For more information about the CAST function, see the *SQL/MP Reference Manual* or online help available through SQLCI.
- A date-time data type cannot be used with other SQL/MP data types except INTERVAL in arithmetic expressions or comparisons. INTERVAL values can be multiplied or divided by scalar data types and added to or subtracted from date-time data types.
- A SIGNED column is required for a number with 10 or more digits.

Performance Considerations

To achieve maximum performance, consider these issues when defining columns:

- Define the column data type so that the values stored in the column match the use of the data in applications. You should attempt to eliminate unnecessary data conversion in programs. Data conversions in programs can decrease application performance.

- Define varying-length columns (VARCHAR, NCHAR VARYING) as the last columns of the table. For all the other data types, the column structure within the table does not affect the performance of queries or updates. For the most efficient use of varying-length columns, however, these columns should be trailing.
- Define columns as numeric if they contain numeric-only values. SQL calculates its execution plan more accurately for numerically defined data.
- Do not define columns as SIGNED numeric unless they need to be signed. Signed columns are less efficient than unsigned columns.
- Place all varying-length variables at the end of a row. If a VARCHAR variable is inside a row, the VARCHAR column is extended to its maximum length, and a second move is required to retrieve any data after the VARCHAR.
- Avoid specifying odd-length strings, such as CHAR (1), CHAR (3), or VARCHAR (5).

  Two moves are required to handle the filler required when an odd-length string precedes a number, INTERVAL, varying-length, or nullable column.
- Define data types to match those used in host variables or by users, or encourage those who use and program the system to match the data types in the database, including date-time data type ranges. This strategy minimizes data type translations. For example, a NUMERIC data type in DDL might translate to a double data type for the host variable in C code. In this instance, you could change the DDL definition to FLOAT(54) so that the two match and do not require translation.

Defining Character Data

SQL includes both fixed-length character data and variable-length character data. The data types for character data are:

CHARACTER NCHAR PIC X DISPLAY	Fixed-length characters
VARCHAR CHAR VARYING NCHAR VARYING	Variable-length characters

Either type of character data can be associated with a character set by specifying the CHARACTER SET clause on the host variable declaration. A character data type is compatible with another character data type with the same character set, but is not compatible with numeric, date-time, or interval data types, and not with character data associated with a different character set.

You can specify one of these character sets for a column:

- ISO 8859/1 through ISO 8859/9: 8-bit character sets, of which ASCII (a 7-bit set) is a subset
- HP Kanji: the HP representation of the character set defined in the JIS X0208 standard and commonly used in Japan
- HP KSC5601: a double-byte character set that is the Korean Industrial Standard character set

SQL/MP also supports the UNKNOWN character set for character columns that do not have a specified character set.

Note. The NATIONAL CHARACTER (or NCHAR) data type uses the default multibyte character set for the node. To use the NCHAR data type, the system default multibyte character set must be a character set that is supported by SQL/MP. For more information, see Hardware and Software Requirements on page 2-1.

Defining Numeric Data

The data types for numeric data are:

NUMERIC PIC 9 COMP	Exact numeric binary data
SMALLINT INTEGER LARGEINT	Binary integer
FLOAT REAL DOUBLE PRECISION	Floating-point number
DECIMAL PICTURE 9 DISPLAY	Decimal numeric ASCII characters

FLOAT is compatible with other numeric data types. SQL/MP performs implicit data conversion from other numeric types to handle arithmetic or comparison operations when required.

A column of an exact numeric type can accept a floating-point number. Also, a column of the FLOAT data type can accept either a floating-point number or an exact numeric type. These rules apply both to columns of an SQL object and to a host variable field.

When SQL/MP performs arithmetic operations on operands that have mixed data types, the data type allowing the largest value is used to evaluate the numbers. For instance, if a REAL number is used and REAL is the data type that allows the largest value, all other numeric data types are converted first to REAL and then used in the expression. These numeric data types are in order of increasing size: DECIMAL, SMALLINT, INTEGER, LARGEINT, REAL, DOUBLE PRECISION.

Defining Date-Time and Time Interval Data

The data types for date-time data are:

DATETIME	Date and time data, optionally including specific time periods
DATE	Date only data
TIME	Time only data
TIMESTAMP	Date and time data

The data type for time interval data is as follows:

INTERVAL	Duration of time

Columns of the DATETIME, DATE, TIME, TIMESTAMP, and INTERVAL data types contain information about dates, times, and time intervals. A comparison of these data types follows, denoting in each case the data type, its meaning, and the range of data allowed in fields of a value of this type. The fields in these data types are not equivalent to columns.

A column value of type DATETIME is made up of any subset of these contiguous fields:

```
YEAR           Year                  1 to 9999
MONTH          Month of year         1 to 12
DAY            Day of month          1 to 31 (maximum value
                                        depends on length of
month)
HOUR           Hour of day           0 to 23
MINUTE         Minute of hour        0 to 59
SECOND         Second of minute      0 to 59
FRACTION       Fraction of second    0 to 0.9(n) in
                                     which n is the precision.
                                     The maximum precision is 6;
                                     the default is 6.
```

A DATETIME column is defined to have a range of these contiguous fields, specified by an optional start-date-time field and a required end-date-time field. You cannot include the FRACTION precision in the start-date-time specification.

A column value of type DATE is made up of these contiguous fields:

```
YEAR            Year                1 to 9999
MONTH           Month of year       1 to 12
DAY             Day of month        1 to 31
```

The DATE data type is equivalent to DATETIME YEAR TO DAY. The maximum value of DAY depends on the length of the month.

An item of type TIME indicates a time of day based on a 24-hour clock. An item of type TIME is made up of these contiguous fields:

`HOUR`	`Hour of day`	`0 to 23`
`MINUTE`	`Minute of hour`	`0 to 59`
`SECOND`	`Second of minute`	`0 to 59`

The TIME data type is equivalent to DATETIME HOUR TO SECOND.

A column value of type TIMESTAMP is made up of any subset of the following year-month or day-time contiguous fields:

`YEAR`	`Year`	`1 to 9999`
`MONTH`	`Month of year`	`1 to 12`
`DAY`	`Day of month`	`1 to 31`
`HOUR`	`Hour of day`	`0 to 23`
`MINUTE`	`Minute of hour`	`0 to 59`
`SECOND`	`Second of minute`	`0 to 59`
`FRACTION`	`Fraction of second`	`0 to 0.9(6), in which 6 is the precision.`

The TIMESTAMP data type is equivalent to DATETIME YEAR TO FRACTION(6). The maximum value of DAY depends on the length of the month.

A column value of type INTERVAL is made up of these contiguous fields:

`YEAR`	`Number of years`	`Not constrained`
`MONTH`	`Number of months`	`0 to 11`
`or`		
`DAY`	`Number of days`	`Not constrained`
`HOUR`	`Number of hours`	`0 to 23`
`MINUTE`	`Number of minutes`	`0 to 59`
`SECOND`	`Number of seconds`	`0 to 59`
`FRACTION`	`Fraction of second`	`0 to 0.9(`*n*`), in which `*n*` is the precision. The maximum precision is 6; the default is 6.`

An INTERVAL column is defined to have a range of these contiguous fields, specified by a required start-date-time field and an optional end-date-time field. You can specify a precision for any start-date-time field, but you cannot include the FRACTION precision in the start-date-time specification.

Guidelines

When you define a column to hold date and time, date, time, or time interval values, use these general guidelines:

- A column of the DATETIME, DATE, TIME, or TIMESTAMP type holds a value that represents a date or an instant in time, and a column of the INTERVAL type holds a value that represents a time interval, or duration.
- A column of the DATE, TIME, or TIMESTAMP type is equivalent to a DATETIME type with a specific range of DATETIME fields.

- A column defined as DATETIME can have a range of DATETIME fields to limit the set of values stored:

```
COLUMN_1    DATETIME YEAR TO SECOND
COLUMN_2    DATETIME MONTH TO DAY
COLUMN_3    DATETIME HOUR TO MINUTE
```

- A date-time column stores values in local civil time (LCT). The LCT is determined by the node where the SQL executor is running, based upon the TIME ZONE OFFSET and DAYLIGHT SAVINGS parameters established for the system during system generation.

- The range of fields defined for an INTERVAL column can limit the value stored; for example:

```
COLUMN_1    INTERVAL YEAR
COLUMN_2    INTERVAL HOUR(3)
COLUMN_3    INTERVAL YEAR TO MONTH
COLUMN_4    INTERVAL DAY TO MINUTE
```

- The fields in a date-time or INTERVAL value have this implied order: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, FRACTION.

- Possible default values for a DATETIME column are:

```
DEFAULT date-time-literal   A valid date-time literal
DEFAULT SYSTEM             Initialized to the current
timestamp
DEFAULT CURRENT            Initialized to the current
timestamp
DEFAULT NULL               Initialized to the null value
```

- Possible default values for an INTERVAL column are:

```
DEFAULT interval-literal    A valid INTERVAL literal
DEFAULT SYSTEM             Initialized to zero
DEFAULT NULL               Initialized to the null value
```

- A date-time column cannot be used with other SQL data types in arithmetic expressions or comparisons, except with INTERVAL data types. This table shows the results of arithmetic operations involving date-time and INTERVAL values:

First Operand	**Operator**	**Second Operator**	**Result**
Date-time	-	Date-time	INTERVAL
Date-time	+ or -	INTERVAL	Date-time
INTERVAL	+	Date-time	Date-time
INTERVAL	+ or -	INTERVAL	INTERVAL
INTERVAL	* or /	Scalar	INTERVAL
Scalar	*	INTERVAL	INTERVAL
INTERVAL	/	INTERVAL	Numeric

- A negative value is not a valid entry for a date-time column. An INTERVAL column, however, can contain negative values.

- A date-time value cannot be multiplied by -1, but an INTERVAL value can be. Negative date-time values are not valid.

These guidelines apply to arithmetic operations on date-time and INTERVAL data types:

- The result of subtracting two date-time values is an INTERVAL value.
- The result of adding or subtracting INTERVAL values is an INTERVAL value.
- The result of adding a positive INTERVAL value to a date-time value is a date-time value with an increased value in a DATETIME field or fields. The result of subtracting a positive INTERVAL value from a date-time value is a date-time value with a decreased value in a DATETIME field or fields.

  For instance, if you add INTERVAL (1) MONTH to a TIMESTAMP value, only the MONTH field changes. The rest of the fields in the TIMESTAMP value remain the same.

  If the INTERVAL value for the field increases or decreases the value so that it affects the value of another field, the other field changes accordingly. For instance, if you add INTERVAL (5) MONTH to a MONTH field that is already 9, both the MONTH field and the YEAR field change. If an arithmetic expression results in invalid data, an SQL error is generated.

Using Default and Null Values

In SQL/MP, a null value is a marker that indicates that a column in a specified row has no value. The null character is not treated as a normal data value; it serves strictly as a placeholder necessary for certain relational operations. To an application interacting with a database, the null value indicates “unknown” or “do not know.”

The DEFAULT and NULL clauses determine the value used when a column value is not supplied for a row during data entry. The following discussion describes the DEFAULT and NULL variations you can use and their effects on the data allowed in columns.

The DEFAULT and NULL clauses are independent of each other and must be specified separately. The options for DEFAULT clause values apply in defining columns as follows:

- Use the NO DEFAULT clause in a column definition when you want your application to explicitly supply values for the column. The NO DEFAULT clause ensures that any inserted or updated row contains a value for the column. The system does not allow the insert or update if the column value is omitted. The NO DEFAULT restriction applies to inserts and updates made either directly to the base table or through protection views. In particular, these guidelines apply:
  - Declaring NO DEFAULT for a column requires the application to supply a value for the column. The supplied value can be a null value.

  - Declaring NO DEFAULT NOT NULL for a column requires the application to supply a value. The supplied value cannot be a null value.

- Use the DEFAULT *literal* clause in a column definition when a literal default value for the column is acceptable. The data type of the literal must match the data type of the column, as follows:
  - Declaring a column with DEFAULT *literal* specifies using the default value if no entry is made for the column. The column can contain a null value.
  - Declaring a column with DEFAULT *literal* NOT NULL specifies using the default value if no entry is made for the column. The column cannot contain a null value.
- Use the DEFAULT SYSTEM clause when you want to use the SQL/MP default value. The column data type determines the assigned system default value. The default value for each type is as follows:

Numeric column	Zero
CHARACTER column	String of blanks
NATIONAL CHARACTER column	String of blanks
VARCHAR column	Zero-length string
NATIONAL CHARACTER VARYING column	Zero-length string
Date-time column	Zero
INTERVAL column	Zero

  Declaring a column with DEFAULT SYSTEM specifies initializing the column to the system default value if no value is supplied for the column. The column can contain a null value.

  Declaring a column with DEFAULT SYSTEM NOT NULL specifies initializing the column to the system default value if no value is supplied for the column. The column cannot contain a null value.
- Use the NOT NULL clause to specify that the column cannot contain null values. If this clause is used separately without a DEFAULT clause, the user must supply a value for this column.
- Use the DEFAULT NULL clause to specify that the column takes on a null value if no value is supplied for it.
- You cannot use the DEFAULT NULL and NOT NULL clauses for the same column.

The use of a column as a key column or the partitioning of a table affects use of the NULL clause as follows:

- A column specified in the PRIMARY KEY or CLUSTERING KEY clause of a CREATE TABLE statement cannot allow null values. Therefore, you should use the NOT NULL clause to define primary key or clustering key columns. In the absence of the NOT NULL clause, any columns defined in the PRIMARY KEY or CLUSTERING KEY clause are implicitly set to NOT NULL, while all other columns are implicitly set to allow nulls by default.

- A column used in nonunique index keys can contain null values because duplicate values are allowed in such columns.
- A column used as a unique key for a single-column index can contain null values, but only one row with a null value is allowed. Therefore, you might want to define a column used in this manner as NOT NULL.

  Unique multicolumn indexes can contain columns with null values. The same rule applies as for single-column unique indexes; that is, the index can have at most one row of all null values in the columns. Null values are treated as all other values and therefore cause duplication of rows in the same manner.
- For partitioning an index, a null value is considered greater than other values and equal to other instances of the null value. You can use the NULL indicator in the FIRST KEY clause to group all rows with null values in the index key into the same partition. In an index with an ascending key, all rows with null values in the key column appear in the last partition; in an index with a descending key, all rows with null values in the key column appear in the first partition.

Specifying Column Attributes

A column definition can specify the HEADING, HELP TEXT, and UPSHIFT attributes, which can be used by programs and retrieved from the catalog description. These descriptive column attributes are used automatically by applications generated through the Pathmaker application development tool. To assist with application development and integrity, the attributes can also be used by applications generated outside of the Pathmaker environment.

HEADING Attribute

The HEADING attribute associates heading text with a column to enable applications to refer to an alternate heading in place of the column name itself. A column can have a default heading of up to 132 characters. Alternatively, if NO HEADING is specified for the column, the application uses the column name as the default.

HELP TEXT Attribute

The HELP TEXT attribute associates help text with a column name to provide help information. The help text is stored in the COMMENTS table, in order of entry, as single rows of up to 77 characters each. You can retrieve the help text by querying the COMMENTS table.

UPSHIFT Attribute

The UPSHIFT attribute is allowed for single-byte character-type columns only. If UPSHIFT is specified, SQL/MP upshifts all data entered before storing the data in the column.

Defining Corresponding Columns in Different Tables

Data type cross-matching between tables is the responsibility of the database administrator. You must ensure that corresponding columns used in different tables are defined with the same characteristics: data type, size, default values, and constraints.

For example, a part-number column defined in several tables should have the same definition. Do not define one column as PIC X(4) and another as PIC 9(4). By using the same column definition, you ensure that applications can perform join operations during data retrieval or predicate comparisons.

Understanding Data Type Correspondence With Host Languages

Data type correspondence between SQL/MP and the host programming languages supported for embedded SQL is shown in Table 5-1. The table lists selected column data types representing different possible data descriptions generated by INVOKE statements.

Table 5-1. Summary of Corresponding Data Types (page 1 of 3)

SQL/MP Column Name and Data Type	COBOL	C	Pascal	TAL
A CHAR(10) PIC X(10) NCHAR (10)	PIC X(10).	char a[11];	fstring(10);	string a[0:9];

B VARCHAR(10)
NCHAR VARYING(10)
	02 B.
	03 LEN PIC S9(4) COMP.
	03 VAL PIC X(10).
		struct {
		short len;
		char val[11];
		}b;
			string(10);	struct b;
				begin
				int len;
				string val[0:9];
				end;

C CHAR(10) CHAR SET y PIC X(10) CHAR SET y NCHAR (10)	01 B CHARACTER SET y PIC X(10).	char CHARACTER SET y a[11];	fstring(10) CHARACTER SET y;	string CHARACTER SET y a[0:9];

D VARCHAR(10) CHAR SET y
NCHAR VARYING(10)
	01 B.
	02 LEN PIC S9(4) COMP.
	02 VAL CHARACTER SET y PIC X(10).
		struct {
		short len;
		char CHARACTER SET y val[11];
		}b;
			string(10) CHARACTER SET y;	struct b;
				begin
				int len;
				string CHARACTER SET y val[0:9];
				end;

Table 5-1. Summary of Corresponding Data Types (page 2 of 3)

SQL/MP Column Name and Data Type	COBOL	C	Pascal	TAL
E NUMERIC(4, 0)	PIC S9(4) COMP.	short c;	int16;	int c;
F NUMERIC(4, 0) UNSIGNED	PIC 9(4) COMP.	unsigned short d;	cardinal;	int d /SMALLINT UNSIGNED/;
G NUMERIC(9, 2)	PIC S9(7)V9(2) COMP.	long e; /* scale is 2 */	int32; {* scale is 2 *}	int(32) e; ! scale is 2
H NUMERIC(9, 2) UNSIGNED	PIC 9(7)V9(2) COMP.	unsigned long f; /* scale is 2 */	int32; {* scale is 2 *}	int(32) f /INTEGER UNSIGNED/; ! scale is 2
I NUMERIC(18, 2)	PIC S9(16)V9(2) COMP.	long long f; /* scale is 2 */	int64; {* scale is 2 *} (No arithmetic in Pascal)	fixed(2) g;
J SMALLINT	PIC S9(4) COMP.	short h;	int16;	int h;
K SMALLINT UNSIGNED	PIC 9(4) COMP.	unsigned short i;	cardinal;	int i /SMALLINT UNSIGNED/;
L INTEGER	PIC S9(9) COMP.	long j;	int32;	int(32) j;
M INTEGER UNSIGNED	PIC 9(9) COMP.	unsigned long k;	{* k: UNSIGNED INTEGER IS NOT SUPPORTE D *}	int(32) k /INTEGER UNSIGNED/;
N LARGEINT	PIC S9(18) COMP.	long long l;	int64; (No arithmetic in Pascal)	fixed(0) l;
O DECIMAL(18, 2)	PIC S9(16)V9(2) DISPLAY SIGN IS LEADING.	decimal m[19]; /* scale is 2 */	decimal(18); {* scale is 2 *}	string m[0:17] /DECIMAL(18)/; ! scale is 2
P DECIMAL(9, 2) UNSIGNED	PIC 9(7)V9(2) DISPLAY.	decimal n[10]; /* scale is 2 */	decimal(9); {* scale is 2 *}	string n[0:8] /DECIMAL(9) UNSIGNED/; ! scale is 2

Table 5-1. Summary of Corresponding Data Types (page 3 of 3)

SQL/MP Column Name and Data Type	COBOL	C	Pascal	TAL
Q PIC 9(9) COMP	PIC 9(9) COMP	unsigned long o;	int32;	int(32) o /INTEGER UNSIGNED/;
R FLOAT(15)	* Q: REAL IS NOT SUPPORTED	float q;	real;	real q;
S FLOAT(30)	* R: DOUBLE PRECISION IS NOT SUPPORTED	double r;	longreal;	real(64) r;
T REAL (Same as FLOAT(22))	* S: REAL IS NOT SUPPORTED	float s;	real;	real s;
U DOUBLE PRECISION (Same as FLOAT(54))	* T: DOUBLE PRECISION IS NOT SUPPORTED	double t;	longreal;	real(64) t;
V DATETIME YEAR TO DAY	PIC X(10).	char u[11];	fstring(10);	string u[0:9];
W DATE (Same as DATETIME YEAR TO DAY)	PIC X(10).	char v[11];	fstring(10);	string v[0:9];
X TIME (Same as DATETIME HOUR TO SECOND)	PIC X(8).	char w[9];	fstring(8);	string w[0:7];
Y TIMESTAMP (Same as DATETIME YEAR TO FRACTION(3))	PIC X(26).	char x[27];	fstring(26);	string x[0:25];
Z INTERVAL MONTH(6)	PIC X(7).	char y[8];	fstring(7);	string y[0:6];

Creating Table Partitions

To promote parallel processing of queries and parallel index maintenance, you should partition data across available disk volumes. For a very large table or a table used at different geographical sites, partitions can make the data more accessible and can reduce the time required for table scans by a factor almost equal to the number of partitions.

Partitions are allowed for tables of all three organization types. For key-sequenced tables, however, the table must have a user-defined primary key to have partitions.

A partition of a table or index holds all the rows within a range of key values. Partitions can reside on one system or across many systems in a network.

You can partition a table or index upon creation. You can also split an existing table or index into partitions, or you can add or drop a partition by using the ALTER TABLE or ALTER INDEX statement. There are special rules for adding, moving, and splitting partitions based on file type (key-sequenced, entry-sequenced, or relative). For more information, see Adding Partitions to Tables and Indexes on page 7-7 and Altering Partition Attributes on page 7-19.

Performance Benefits

The benefits of using partitioned tables and indexes are:

- Partitions are independent of one another for access. Only the accessed partition must be available.
- Partitions improve transaction throughput by allowing simultaneous disk access to different partitions of the same table.
- Partitions require no special access procedures. SQL manages partition access for you automatically.
- Partitions enable SQL to more readily process queries in parallel.
- Partitions allow you to have tables larger than the size of a single disk volume.

Partitioning the indexes of a table enables SQL to take maximum advantage of parallel index updates. Indexes should reside on separate volumes and should be configured on separate processors.

Creating Partitions on a System That Uses SMF

If you create tables and indexes with a large number of partitions—for example, tables supporting a DSS application—you can use the SMF product to manage the distribution of partitions across physical disk volumes. When you create a table or index, you can specify partition names using virtual volumes as the volume portion of the *`volume.subvolume.file`* name; SMF automatically distributes the partitions across the physical volumes assigned to the storage pool (or pools). You can also place nonpartitioned tables and indexes on virtual volumes managed by SMF.

examples suggest possible configurations of storage pools:

- You can configure a pool containing physical volumes primaried to a particular processor or set of processors. By partitioning a table across virtual volumes associated with that pool, you ensure that disk access to the table will be managed by the specified processors.

  SQL/MP provides a complementary mechanism for controlling (limiting) which processors run parallel queries. For more information about using SMF with this feature, see Managing Processor Usage in a Distributed Environment on page 12-11.

- You can configure a pool for a particular application. Create SQL objects for that application on virtual volumes associated with the pool; the objects will reside on the physical volumes in the pool. To maximize performance of parallel queries, you can balance the number of physical volumes primaried to each processor.

- As a variation on the preceding configurations, you can configure a pool for each processor. That is, the physical volumes in each pool are primaried to a particular processor. You then create partitions on virtual volumes associated with each pool; be sure to distribute the partitions across all the pools. This configuration allows SMF to manage the physical storage of files and ensures that the partitions are distributed across all processors, which enhances the parallel execution of queries.

- However you partition tables and indexes, you can allow temporary files used during SQL queries (such as files used for repartitioning or FastSort scratch files) to be distributed across physical volumes primaried to all processors.

The preceding examples are general suggestions for the use of SMF. You will need to configure partitions and storage pools according to the requirements of your environment. For more information about SMF, see the *Storage Management Foundation User’s Guide*.

SMF allows multiple virtual volumes to be associated with the same storage pool, which can cause the storage of two or more partitions of the same file being allocated to the same physical disk, if that is determined to be the most efficient use of available space. To ensure that a partition resides on a unique physical volume, you can place it on a direct volume that is not in any storage pool or use the PHYSVOL option when you create or alter a table or index.

If a direct volume is also in a storage pool, it becomes more complicated to ensure that a specified partition is the only one residing on that physical volume. To achieve this goal, it is generally better to specify a virtual volume for the partition and use the PHYSVOL option than to specify a direct volume. When you use the PHYSVOL option, the partition has the benefit of being managed by SMF.

The PHYSVOL option lets you place a partition on a specified physical volume in the pool. You do not have to use the PHYSVOL option on all partitions in a table. You can use it for particular partitions that you want to locate on specified physical disks, while letting SMF manage the physical location of other partitions in the table.

The trade-off of using PHYSVOL is that when you assign a partition to a particular physical volume, SMF might not be able to relocate the file (partition) automatically. For the syntax and definition of the PHYSVOL option, see the CREATE and ALTER entries in the *SQL/MP Reference Manual*.

Defining a Large Number of Partitions

To take advantage of the larger number of indexes and the larger number of partitions available for tables and indexes on versions 320 and later of SQL software, use the PARTITION ARRAY EXTENDED option when you create the base table.

The length of the key is a factor in determining the maximum number of partitions for a table or index and relates to the storage of FIRST KEY values for all the partitions in a fixed-size file label; however, the actual calculation is complex and depends on additional factors, such as disk label space and the message size of the operating system. Examples of partition limits for base tables are:

Primary Key Size (bytes)	**Approximate Limit on Number of Partitions (with EXTENDED partition array)**
10	900
44	450
100	250
250	110

Examples of partition limits for indexes are:

Primary Key Size (bytes)	**Index Key Size (bytes)**	**Approximate Limit on Number of Partitions (with EXTENDED partition array)**
10	10	670
10	32	450
50	48	250
100	100	130

Note that the size of the primary key of an index table equals (primary key of the base table + index key + 2 bytes). The maximum number of partitions for a base table with a primary key length of 44 bytes is the same as that of an index table with a primary key of 10 bytes, an index key of 32 bytes, and the 2-byte offset.

Note. SQL tables and indexes with many partitions (typically around 400) might cause SQLCAT, SQLUTIL, or AUDSERV processes to incur file-system error 31 or 34 because of insufficient memory in the process file segment (PFS). To increase the PFS size for any of these SQL processes, use the BIND statement CHANGE PFS command. For programs run from TACL, you can specify the PFS size in the TACL RUN command. Save the original copy of any program you modify. If you alter PFS size for SQLCAT, SQLUTIL, or AUDSERV, you must license the modified copy and re-alter the PFS size when you install a newer version of SQL/MP.
SQL tables and indexes with many partitions might also cause the PARTNS catalog table and its associated index, IXPART01, to become full. The PARTNS table contains N**2 rows for each table and index with N partitions. Thus, three tables of 400 partitions each would fill the PARTNS table. To remedy this situation, distribute the definitions of objects across catalogs.

Special Considerations for DSS Applications

DSS applications typically require periodic addition and deletion of data. Further, access must be well balanced. The choice of leftmost primary key column—and thus the partitioning strategy—can greatly affect access. Choose a leftmost key column that minimizes contention while allowing acceptable load and delete performance. Possibilities include a date column (if access is distributed evenly across date values), a nondate column in the primary key (if access across the date column is weighted toward specific ranges), or an artificial “partition number” value, with rows distributed across all partitions.

Further, when defining partitions for a DSS application, consider using one or more of these strategies:

- Leave the primary partition empty (with minimal size) and use secondary partitions for all data. The primary partition of a partitioned table cannot easily be dropped. If all your data resides in secondary partitions, you have the greatest flexibility for adding, dropping, and splitting partitions.
- Mirror the volume associated with the primary partition to maximize availability. Queries and programs that refer to the primary partition require its availability when accessing data.
- Configure mirrored disks on separate channels. This strategy maximizes performance and should be used for volumes containing primary and secondary partitions where possible.

Sample Table Definition

This example shows the definition for the PARTLOC table. This table contains data maintained at three different sites: New York, Los Angeles, and San Francisco. The table is partitioned by its primary key (which consists of LOC_CODE and PARTNUM) into three partitions, one for each site:

```
CREATE TABLE \NY.$WHS1.INVENT.PARTLOC (
    LOC_CODE       CHARACTER (3)         NO DEFAULT,
    PARTNUM        NUMERIC (4) UNSIGNED  NO DEFAULT,
    QTY_ON_HAND    NUMERIC (7)           NO DEFAULT,
    PRIMARY KEY (LOC_CODE, PARTNUM
)

CATALOG \NY.$WHS1.INVENT
PARTITION (\SF.$WHS2.INVENT.PARTLOC
            CATALOG \SF.$WHS2.INVENT
            FIRST KEY ("G00", 0) ,
           \LA.$WHS3.INVENT.PARTLOC
            CATALOG \LA.$WHS3.INVENT
            FIRST KEY ("P00", 0)
           ) ;
```

This example creates a partitioned table with a primary partition and two secondary partitions. The example shows the use of DEFINE names to identify a table and catalog.

```
-- DEFINEs were previously added during this SQLCI
-- session or inherited from the command interpreter.

>> INFO DEFINE =MCAT;
     DEFINE NAME             =MCAT
     CLASS                   CATALOG
     SUBVOL                  \SYS1.$VOL1.MFG

>> INFO DEFINE =ORDERS;
     DEFINE NAME             =ORDERS
     CLASS                   MAP
     FILE                    $VOL1.MFG.ORDERS

>>  CREATE TABLE  =ORDERS
+>    (ORDERNUM    DECIMAL (6) UNSIGNED  NO DEFAULT NOT NULL,
+>     ORDER_DATE  DATETIME YEAR TO DAY  NO DEFAULT NOT NULL,
+>     DELIV_DATE  DATETIME YEAR TO DAY  NO DEFAULT NOT NULL,
+>     SALESREP    DECIMAL (4) UNSIGNED  DEFAULT SYSTEM,
+>     CUSTNUM     DECIMAL (4) UNSIGNED  NO DEFAULT NOT NULL,
+>      PRIMARY KEY ORDERNUM)
+>      PARTITION (=ORDERS2
+>                 CATALOG =MCAT
+>                 FIRST KEY 20000,
+>                =ORDERS3
+>                 CATALOG =MCAT
+>                 FIRST KEY 40000)
+>     EXTENT (1000,100)
+>     BLOCKSIZE  2048
+>     CATALOG =MCAT
+>     SECURE "NNOO";
--- SQL operation complete.
```

For more information, see the *SQL/MP Reference Manual* or the SQLCI HELP entry for the CREATE TABLE statement

Securing a Base Table

Base tables are the foundation of the database, and base table security ultimately defines much of the security for views, indexes, and DML statements. The local owner of an object, a remote owner with purge authority, and the super ID user generally have the authority to perform DDL operations on existing tables. Anyone with authority to purge a table can drop the table.

You can alter the security of a table by using the ALTER TABLE statement or the SECURE command.

Security of Dependent Objects

When you alter the security of a base table, SQL/MP automatically alters the security attributes for the dependent indexes. The security of dependent protection views might also be altered if the new security of the table violates the system-enforced relationship between these objects, as explained later in “Security Guidelines for Protection Views.”

Examples of Securing a Table

These examples show ways of securing a table to control DML access and to control who has the authority to perform this set of DDL operations. Authority to purge the table and ownership of the table is required for any of these operations:

- Create a protection view
- Create or drop a constraint
- Create or drop an index
- Update statistics for a table
- Alter the attributes
- Add a partition to a table
- Add a column to a table

This example shows security that enables any network user to read or write to a table. Only the product manager (user 200, 255) can perform the listed DDL operations.

```
$VOL2.APPLTAB.TABLE1     Owner    = 200,255
                         Security = "NN-O"
```

This example shows security that enables any network user to read a table but restricts update operations to the local application group. Only the application manager (user 250, 255) has the authority to perform the listed DDL operations.

```
$VOL2.APPLTAB.TABLE2     Owner    = 250,255
                         Security = "NGOO"
```

This example shows security that enables any network user to read or write to a table. Only the development manager (user 100, 255) has the authority to perform the listed DDL operations. Although the security gives read, write, and purge access to additional users, the DDL statements, except DROP TABLE, require ownership of the table. Any user in group 100 can drop the table.

```
$VOL2.APPLTAB.TABLE2     Owner    = 100,255
                         Security = "NNOG"
```

Creating Views of Base Tables

To create a view, use the CREATE VIEW statement. CREATE VIEW statements enable you to define new column names for a view, instead of using the column names from the underlying tables. When using the view, applications use the view-defined column names. Applications can also use views to rename, reorder, and project subsets of columns from one or more base tables. For background information about views, see Using Views on page 3-13.

Before you create views for your database, see the *SQL/MP Query Guide* for information about formulating queries.

Creating a Protection View

A protection view is a view that has only one underlying table and is defined with the PROTECTION attribute. The view can be a projection of columns or a selection of rows from the underlying table, or both. The rows in a protection view are accessible by applications for read, insert, update, and delete operations. A protection view can be secured for read, write, execute, and purge access.

If you plan to use similarity checking with a protection view, be sure to use the CREATE VIEW statement’s SIMILARITY CHECK ENABLE clause.

Security Guidelines for Protection Views

These guidelines apply to securing protection views:

- A protection view can have only a single underlying table. You must put a protection view on the same volume as its underlying table and register the view in the same catalog as the table.
- To create a protection view, you must have read, write, and purge authority for the underlying table and write authority for the catalog in which the underlying table is registered. You must also be the local owner of the underlying table or a remote owner with authority to purge the table.
- The owner of a protection view is always the same as the owner of the underlying table.
- The security string of a protection view has these dependencies on the underlying table:

- The purge authority for the protection view must include all users who have purge authority for the underlying table. Normally, the purge authority for a protection view is the same as the purge authority for the underlying table.
- The security string at creation time must meet security dependency requirements. If the creator's default security string violates the rules, the creation attempt fails. The creator can then include the SECURE clause in the CREATE VIEW statement and reissue the statement.
- If an alteration of the security string for the underlying table violates the security criteria, SQL issues a warning and automatically changes the security strings for the protection view.

Examples

This example creates a protection view of the EMPLOYEE table. The table columns are renamed in the view definition.

```
>>  CREATE VIEW $VOL1.PERSNL.EMPLIST
+>      (EMPLOYEE_NUMBER, LAST_NAME, FIRST_NAME, DEPARTMENT)
+>      AS SELECT EMPNUM, LAST_NAME, FIRST_NAME, DEPTNUM
+>      FROM EMPLOYEE
+>      FOR PROTECTION
+>      WITH HEADINGS
+>      CATALOG $VOL1.PERSNL;
```

This example shows a view definition including the WITH CHECK option and the effect on an attempted insertion of a row that violates the WITH CHECK condition. Only rows with EMPNUM values greater than 1 and less than 1000 can be inserted.

```
>>CREATE VIEW $VOL1.PERSNL.EMPVIEW
+>    AS SELECT EMPNUM, FIRST_NAME, LAST_NAME,
+>              DEPTNUM, JOBCODE, SALARY
+>    FROM $VOL1.PERSNL.EMPLOYEE
+>    WHERE EMPNUM > 1 AND
+>          EMPNUM < 1000
+>    FOR PROTECTION
+>    CATALOG $VOL1.PERSNL
+>    SECURE "NCNC"
+>    WITH CHECK OPTION;
--- SQL operation complete.

-- Because the WITH CHECK OPTION is defined, the following
-- INSERT statement is rejected:

>>INSERT INTO $VOL1.PERSNL.EMPVIEW (*)
+>   VALUES (1200, "WAYLAN","JAMES", 4000, 250, 55000.00 );
                                                          ^
*** ERROR from SQL [-8300]: File system error occurred on
          \SYS1.$VOL1.PERSNL.EMPVIEW.
*** ERROR from File System [1026]:  The selection expression
      on an SQL view has been violated.
```

Creating a Shorthand View

A shorthand view is a view derived from one or more tables and other views and defined without the PROTECTION attribute. A shorthand view can only be queried and it can be secured only for purge access. Any user who has authority to read the underlying table or tables, has authority to read the shorthand view.

Security Guidelines for Shorthand Views

- A shorthand view can have more than one underlying table or view or a combination of underlying tables and views.
- The view's creator must have authority to write to the VIEWS catalog table that contains the view description and to the USAGES and TRANSIDS catalog tables in the catalogs that contain the descriptions of the underlying tables and views.
- The owner of a shorthand view is set to that of the creator. The security string defaults to the current default security of the creating process. You can specify a security string in the SECURE clause of the CREATE VIEW statement.
- The owner of a shorthand view does not have to be the same as the owner of an underlying table or view. Different users can own the tables and views underlying a shorthand view.
- Only purge authority has meaning in the security string for a shorthand view, although the entire string is required. Anyone with authority to read all the underlying tables has authority to read a shorthand view. When you create a shorthand view, you should verify that read authority is available for all the underlying tables and views.

Examples

This example creates a shorthand view of two tables by using the DEPTNUM columns in the joining criterion. Correlation names are required because DEPTNUM appears in both tables.

```
>>  CREATE VIEW \SYS1.$VOL1.PERSNL.EMPDEPT
+>      AS SELECT E.EMPNUM, E.LAST_NAME, E.FIRST_NAME,
+>                E.DEPTNUM, D.DEPTNAME
+>      FROM \SYS1.$VOL1.PERSNL.EMPLOYEE E,
+>           \SYS1.$VOL1.PERSNL.DEPT D
+>      WHERE E.DEPTNUM  = D.DEPTNUM
+>      CATALOG \SYS1.$VOL1.PERSNL;
```

Example 5-2 on page 5-41 creates a shorthand view of four joined tables. Correlation names are required because of duplicate column names in the tables. The INFO DEFINE command displays the DEFINEs used in creating the view. For more information about DEFINEs, see Using DEFINEs on page 10-30.

Example 5-2. A Shorthand View of Four Joined Tables

```
-- DEFINEs were previously added during this SQLCI
-- session or inherited from the command interpreter.
>> INFO DEFINE =ICAT;
     DEFINE NAME            =ICAT
     CLASS                  CATALOG
     SUBVOL                 \SYS1.$VOL3.INVT
>> INFO DEFINE =INVENTORY_VIEW1;
     DEFINE NAME            =INVENTORY_VIEW1
     CLASS                  MAP
     FILE                   $VOL1.MFG.IVIEW1
>> INFO DEFINE =PARTS;
     DEFINE NAME            =PARTS
     CLASS                  MAP
     FILE                   $VOL1.MFG.PARTFILE
>> INFO DEFINE =ORDERS;
     DEFINE NAME            =ORDERS
     CLASS                  MAP
     FILE                   $VOL2.MFG.ORDFILE
>> INFO DEFINE =ODETAIL;
     DEFINE NAME            =ODETAIL
     CLASS                  MAP
     FILE                   $VOL2.MFG.ODETFILE
>> INFO DEFINE =PSUPPLIER;
     DEFINE NAME            =PSUPPLIER
     CLASS                  MAP
     FILE                   $VOL3.MFG.PSUPFILE
>>  CREATE VIEW =INVENTORY_VIEW1
+>      AS SELECT A.PARTNUM, A.PARTDESC, A.PRICE, D.PARTCOST,
+>                C.QTY_ORDERED, A.QTY_AVAILABLE,
D.QTY_RECEIVED,
+>                B.DELIV_DATE,D.EST_RECD_DATE
+>      FROM =PARTS A, =ORDERS B, =ODETAIL C, =PSUPPLIER D
+>      WHERE A.PARTNUM  = C.PARTNUM AND
+>            A.PARTNUM  = D.PARTNUM AND
+>            C.ORDERNUM = B.ORDERNUM
+>      CATALOG =ICAT;
```

View Security and Underlying Table Security

These examples show how the security of a view depends on its underlying table or tables. Suppose that the underlying table has this security:

```
$VOL4.APPLTAB.EMPLOYEE   Owner    = 200,255
                         Security = "GG-O"
```

This example shows a protection view on the table $VOL4.APPLTAB.EMPLOYEE. The owner of the underlying table is the same as the owner of this view. The security of the view enables any group 200 user to read the view but restricts updating and inserting rows to the view owner.

```
$VOL4.APPLTAB.PREMPV1    Owner    = 200,255
                         Security = "GO-O"
```

This example shows a protection view on the table $VOL4.APPLTAB.EMPLOYEE. The owner of the underlying table is the same as the owner of this view. The security of the view enables any network user to read the view and any local group 200 user to update or insert rows into the view.

```
$VOL3.APPLTAB.PREMPV2    Owner    = 200,255
                         Security = "NG-O"
```

This example shows a shorthand view on the table $VOL4.APPLTAB.EMPLOYEE and other tables. The owner of the view can be different from the owner of the underlying tables. The security of the shorthand view is not the basis for access. Read access to the underlying tables authorizes access to this view. For this example, only group 200 users can read the view because $VOL4.APPLTAB.EMPLOYEE is secured for group 200 users.

```
$VOL1.APPLTAB.SHEMPV1    Owner    = 100,255
                         Security = "NNNO"
```

Creating Indexes on Base Tables

An index provides an alternate access path to a table; the alternate path is different from the inherent access path (primary key). Indexes can improve application performance for data retrieval operations by providing the optimizer with a greater choice of access paths. Indexes can be scanned forward or backward.

If an existing index includes the selection columns for an SQL statement, the SQL compiler uses the index as an access path to the data. For more information about indexes and performance, see Determining When to Use Indexes on page 3-16, Optimizing Index Use on page 14-16, and Maximizing Parallel Index Maintenance on page 14-17.

Creating an Index

To create an index, use the CREATE INDEX statement, which creates both the index definition in the catalog and the physical file. If the underlying table contains data, the creation process automatically loads the index. CREATE INDEX statements refer to existing columns of a base table and create alternate indexes on the specified columns.

When you define an alternate index, first consider the column-related guidelines described under Defining Columns on page 5-19.

Also consider primary key definitions, as noted in Primary Keys on page 3-2 and Creating Key-Sequenced Tables on page 5-11. Determine if the primary key is the most appropriate based on actual use of the table.

Guidelines

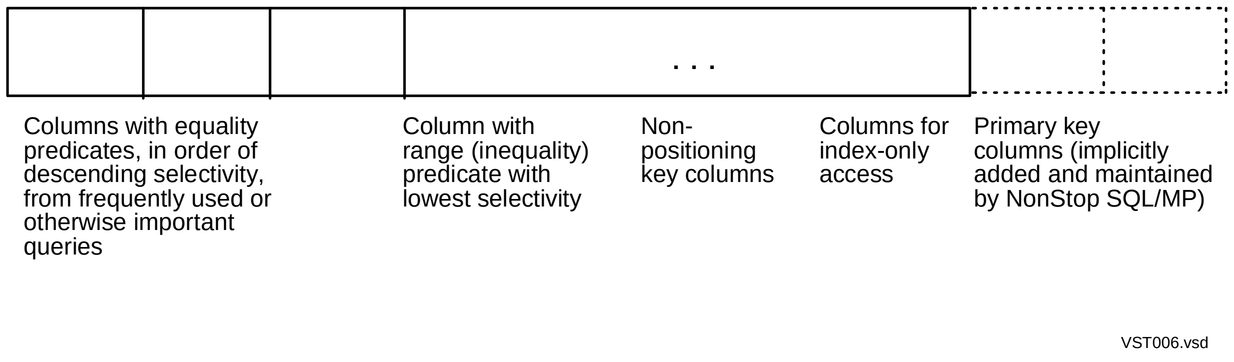
When defining indexes, consider these general guidelines:

- When you create an index, you specify the column or columns that make up the key. The key is classified into one of three levels, depending on the data types of the columns that make up the key. The key level affects performance as described under Key Levels on page 3-4.
- When you create an index, the index inherits the type of partition array associated with the base table. To take advantage of the larger number of indexes and index partitions available with versions 320 and later of SQL/MP software, specify an extended partition array for the base table.

Note. SQL tables and indexes with many partitions (typically around 400) might cause SQLCAT, SQLUTIL, or AUDSERV processes to incur file-system error 31 or 34 or cause the PARTNS catalog table and its associated index, IXPART01, to become full. For more information about this situation, see Creating Table Partitions on page 5-32.

- Define key columns with NO DEFAULT NOT NULL and define nonkey columns with SYSTEM DEFAULT NOT NULL (if the columns do not allow null values) to save two bytes of storage space for each column and avoid complex null logic.
- Define index columns so that their order reflects frequency of access and relative importance of queries competing for the index (see Figure 5-2 on page 5-44):
    - Specify columns to be accessed with equality predicates as leftmost columns.
    - Follow these columns by columns with inequality predicates (predicates that specify a range).
    - If there is more than one column that will be accessed with an inequality condition, consider ordering the columns by decreasing selectivity (as described in the *SQL/MP Query Guide*). These columns become positioning columns because they are used to position within the index and mark the range of qualifying rows.
    - Follow the index columns with nonpositioning columns that will have predicates specified on them and columns that will be selected and have not been included already.

Figure 5-2. Ordering Columns Within an Index


. . .
Columns with equality predicates, in order of descending selectivity, from frequently used or otherwise important queries
Column with range (inequality) predicate with lowest selectivity
Non-positioning key columns
Columns for index-only access
Primary key columns (implicitly added and maintained by NonStop SQL/MP)
VST006.vsd

- Arrange columns to minimize unpacking or maximize bulk move operations, depending on how the columns are used:
  - To avoid unpacking operations, arrange columns to end at word boundaries.
  - To optimize bulk moves, place columns in groups that will be selected together.
- An index column can refer to a different collation from the collation used by the corresponding base table column, provided the shifting rules for both collations are the same.
- Collation of single-byte character data is performed in the order represented by the ordinal positions of the characters in the ASCII set. Alternatively, you can specify a different collating sequence for single-byte character columns by creating a collation object and associating the collation with the character data. If you do associate a collation with the column, the character set associated with the collation must be the same as the character set associated with the column.
- Indexes you create before a table is loaded are loaded automatically as the table is loaded.
- You can load partitioned indexes in parallel. For more information, see Specifying Parallel Loading of Index Partitions on page 5-49.
- If you are loading an index on a large table, you might need to set the =_SORT_DEFAULTS DEFINE to enable FastSort to use alternate swap files or scratch file volumes. The *SQL/MP Reference Manual* describes this DEFINE.
- Indexes can be updated in parallel by the disk process after the table has been updated. To take full advantage of parallel updating, you should create a table's indexes on separate disk volumes, with each disk volume configured for a separate processor. The performance effects of parallel updates are discussed under Maximizing Parallel Index Maintenance on page 14-17.
- If you are creating an index on an existing table, follow the CREATE INDEX statement with an UPDATE STATISTICS statement to update the statistics in the catalog for the table.

- Index creation can be a long operation, depending on the size of the table and the load on the system. Therefore, two locking strategies are available:
  - Default locking requires a shared table lock on the underlying table. The shared table lock ensures that no users can modify rows during the creation of the index. This lock can prohibit access to the table by other users.
  - The WITH SHARED ACCESS option for the CREATE INDEX statement allows access to the table for DML operations during all but the short final stage of index creation. The option includes a reporting feature for monitoring index creation. In addition, you can request a time window or request explicit operator authorization for the final stage of index creation that requires table locking.
  - The WITH SHARED ACCESS option can be used in conjunction with the PARALLEL EXECUTION ON option if the initiating node, all nodes with base table partitions, and all nodes that will have index partitions are running version 315 or later of SQL/MP software.

  For more information about the WITH SHARED ACCESS option and concurrent access to tables by multiple users, see Understanding the Implications of Concurrency on page 14-1 and the “WITH SHARED ACCESS” description in the *SQL/MP Reference Manual*.
- For an audited index, make a TMF online dump of the index immediately after creating it to prepare for possible file recovery, which might be faster than rebuilding the index. The WITH SHARED ACCESS option, when used with the CREATE INDEX statement, allows you to take the online dump while the CREATE INDEX operation is progressing.
- Creating an index invalidates any registered programs that use the underlying table unless you use one of these:
  - The CHECK INOPERABLE PLANS compiler option, described in Using Similarity Checks on page 10-15, with similarity checking enabled for the index. The use of similarity checking is the recommended method for avoiding automatic recompilations.
  - The NO INVALIDATE option in the CREATE INDEX statement

  An invalidated program can be executable, but you must explicitly SQL compile the program to revalidate it. If you do not use similarity checking, or if the similarity check does not succeed, you must recompile to avoid automatic recompilation.
- You can influence the optimizer’s choice of index by using the CONTROL QUERY directive. For more information, see the *SQL/MP Query Guide* and the *SQL/MP Reference Manual*.
- If you are running SQL/MP on a system using the SMF product, you can specify a virtual volume for the index. The virtual volume is associated with a storage pool; SMF places the index file on a physical volume in that storage pool. SMF chooses

a physical location based on its size estimate of the index file and on the available space in the pool.

In exceptional cases you might want a file to reside on a particular physical volume. To accomplish this, you can specify a direct volume that is not in any storage pool, or you can use the PHYSVOL option to specify a particular physical volume in a pool. For more information about using this feature, see the *SQL/MP Reference Manual* and the *Storage Management Foundation User's Guide*.

Examples

This example creates an index on the ORDERS table by using the CUSTNUM column as the key:

```
>> CREATE INDEX $VOL1.SALES.XORDCUS
+>     ON $VOL1.SALES.ORDERS (CUSTNUM)
+>     KEYTAG "OC"
+>     EXTENT (100,50)
+>     BLOCKSIZE  2048
+>     MAXEXTENTS 24
+>     ICOMPRESS
+>     CATALOG $VOL1.SALES;
--- SQL operation complete.
```

This example creates an index on the ORDERS table in order of the most recent DELIV_DATE:

```
>> CREATE INDEX $VOL1.SALES.XORDCUS
+>     ON $VOL1.SALES.ORDERS (DELIV_DATE DESC)
+>     KEYTAG "LD"
+>     EXTENT (100,100)
+>     ICOMPRESS
+>     CATALOG $VOL1.SALES;
--- SQL operation complete.
```

This example creates an index on a virtual volume, $VIR1, associated with a SMF pool. The example specifies that the file reside on the physical volume, $PVOL6:

```
>>  CREATE INDEX $VIR1.SALES.XORDCUS
+>     ON $VIR1.SALES.ORDERS (CUSTNUM)
+>     |
+>     CATALOG $VIR1.SALES
+>     PHYSVOL $PVOL6;
--- SQL operation complete.
```

In the preceding example, $VIR1 must be associated with the storage pool that contains $PVOL3. When you create an index name on a virtual volume such as $VIR1, you use the same syntax you would use for volumes not managed by SMF. However, with the PHYSVOL option, you specify only the volume name (such as $VOL6). SQL returns an error if you specify a full *`volume.subvolume.file`* name with the PHYSVOL option. If you omit the PHYSVOL option, SMF determines the physical volume on which the XORDCUS index resides.

Defining Unique Indexes

You can use the technique of defining index keys as UNIQUE to enhance the performance of a SELECT statement that returns only one row. A unique index requires that the value of the columns that make up the index key is unique in the table. The index value is the value of columns together in the index and not the individual values of the column.

If the table contains duplicate values in the specified columns, you cannot create a unique index on those columns.

You should consider creating a unique index if you know that a column contains only unique values and the column is used in SELECT queries that have a WHERE clause of this format:

```
WHERE column = expression
```

If you create a unique index on the column, the system determines that only one value can be returned, and the execution of the statement requests only one row.

If there is no unique index on the column, the system expects duplicate values in the column and the executor must request a scan of values from the disk process. The system then requests the disk process to validate that only one row satisfies the WHERE clause of the statement.

This example shows a SELECT statement that uses the column SOC-SECURITY-NUMBER, which has been defined as a unique index. The SQL executor requests only one row from the disk process.

```
>> SELECT LAST_NAME, FIRST_NAME, EMPNUM, SOC-SECURITY-NUMBER
+>   FROM EMPLOYEE
+>   WHERE SOC-SECURITY-NUMBER = "534-90-1111";
```

Defining a unique index on columns is a technique that you might use to improve the response only for the SELECT queries that use the column or columns indexed and that return only one row. As with all indexes and keys, you must consider performance issues for all DML operations before determining when to create an index.

For more information on how indexes can enhance performance, see Determining When to Use Indexes on page 3-16, Optimizing Index Use on page 14-16, and Maximizing Parallel Index Maintenance on page 14-17.

Creating Unique Indexes for Integrity Checking

One use of a unique index is to provide an integrity check between rows that is not possible by using constraints. An index defined as unique prevents two or more rows of the table from having the same values in the indexed columns. The system ensures that rows inserted or updated satisfy the unique index requirement.

This example creates a unique index on the EMPLOYEE table. Each row must contain a unique value (a social security number, in this case) because the index is unique.

```
>>  CREATE UNIQUE INDEX IEMPSS ON $VOL1.PERSNL.EMPLOYEE
+>      (EMP_SOCIAL_SEC_NUMBER  ASC)
+>      CATALOG \SYS1.$VOL1.ADMIN;
--- SQL operation complete.
```

If the index definition specifies multiple columns, the value of the columns as a group instead of the values of the individual columns determines uniqueness.

You can make all columns unique by creating a unique index for each column of the base table.

A collation specified for an index column can affect the uniqueness of rows. Rows that contain unique values in a base table column that uses one collation might not contain unique values for an index column that uses a different collation. The uniqueness is based on collation rules.

Creating Index Partitions

The following CREATE INDEX statement creates a partitioned index with the primary partition and an additional partition. The INFO DEFINE command displays the DEFINEs used to identify the index, table, and catalog.

```
-- DEFINEs were previously added during this SQLCI
-- session or inherited from the command interpreter.
>> INFO DEFINE =PCAT;
     DEFINE NAME            =PCAT
     CLASS                  CATALOG
     SUBVOL                 \SYS1.$VOL1.PERSNL

>> INFO DEFINE =EMPFILE;
     DEFINE NAME            =EMPFILE
     CLASS                  MAP
     FILE                   $VOL1.PERSNL.EMPLOYEE
>> INFO DEFINE =EMP_NAME_INDEX;
     DEFINE NAME            =EMP_NAME_INDEX
     CLASS                  MAP
     FILE                   $VOL1.PERSNL.XEMPNAME
>>  CREATE INDEX  =EMP_NAME_INDEX
+>      ON =EMPFILE  (LAST_NAME, FIRST_NAME)
+>      PARTITION ($VOL2.PERSNL.XEMPNAME
+>                  CATALOG =PCAT
+>                  FIRST KEY "M")
+>      EXTENT (1000,100)
+>      BLOCKSIZE  2048
+>      CATALOG =PCAT
+>      SECURE "NNOO";
--- SQL operation complete.
```

Note that an index inherits the partition array value associated with its base table.

For performance considerations see Creating Table Partitions on page 5-32.

Specifying Parallel Loading of Index Partitions

To load the partitions of a partitioned index in parallel, use the PARALLEL EXECUTION ON option of the CREATE INDEX statement or the LOAD command. The parallel feature loads all partitions of an index at the same time by using multiple processes in parallel. The parallel feature does not load more than one index at the same time.

For example, this statement specifies parallel processing for loading index partitions:

```
>> CREATE INDEX AGEINDEX ON CUSTABLE PARALLEL EXECUTION ON
+>   PARTITION ( $VOL2.CUSTOMER.AGEINDEX FIRST KEY 36 ) ;
```

To load the index partitions in parallel, SQL starts processes called record generators and sort processes. The record generators read the base table; the sort processes sort rows and write them to the index. SQL creates one record generator process for each partition of the base table and one sort process for each partition of the index. If the base table is not partitioned, the RDBMS creates only one record generator process and one sort process.

If the target base table in a LOAD command has more than one partitioned index, then the partitions of the first index are loaded in parallel. After the first index has been loaded, the partitions of the second index are loaded in parallel, and so forth.

Performance Considerations

Although the processor cycles and disk processes used in parallel processing might be approximately equal to the processor cycles and disk processes used in serial (nonparallel) processing, parallel processing uses more of these cycles and processes at the same time and might temporarily monopolize system resources.

For best performance, the disk processes for the volumes used should be distributed evenly across all processors. If the index is partitioned, the processor for the volume's disk process should be available for loading each partition.

By default, parallel index loading uses SORTPROG and RECGEN processes located on the nodes where the partitions reside. When the total number of table and index partitions nears 750, however, a load operation might stop with an SQL error 1910 and a sort start error 10, or with an SQL error 1928, record generator error 10. When any of these errors occur, increase the process file segment (PFS) space of the SQLCAT process by using the BIND statement SET PFS command. Alternatively, for programs run from TACL, you can specify the PFS size in the TACL RUN command. Save the original copy of SQLCAT and license the new copy.

Using a Configuration File

You can specify the processors and other configuration options for both record generators and sort processes in a configuration file. In a LOAD command, you can specify a different configuration file for each partitioned index. The configuration options you can specify include:

- Default priority for the record generators and the sort processes (PRI)
- Default object files for the record generators and the sort processes (PROGRAM)
- Default number of records (NUMRECS)
- Default pool of processors in which to run the record generators and another pool in which to run the sort processes (processor)
- Default pool of volumes to use for the initial set of sort scratch files for the sort processes (SCRATCH)
- Default pool of volumes to use for overflow storage for the sort processes if needed (SCRATCHON)
- Set of volumes to exclude from overflow storage (NOSCRATCHON)
- Default pool of volumes to use for swap files for the record generators and another pool for swap files for the sort processes (SWAP)

In addition, you can specify any of these attributes for a specific partition.

Example 5-3 on page 5-51 shows a sample configuration file:

Example 5-3. Sample Configuration File

```
==  Sample configuration file for loading index
==  partitions in parallel. Creates index AGEINDEX
==  on table CUST, which is partitioned as follows:
==     $DATA1.SALES.CUST
==     $DATA2.SALES.CUST
==     $DATA3.SALES.CUST
==     \NEWYORK.$DATA1.SALES.CUST

==  AGEINDEX is partitioned as follows:
==     $DATA4.SALES.AGEINDEX
==     $DATA5.SALES.AGEINDEX
==     \NEWYORK.$DATA2.SALES.AGEINDEX
==     \NEWYORK.$DATA3.SALES.AGEINDEX
==  Set up a default priority for the RECGEN processes.
CREATEINDEX BASETABLE DEFAULT PRI ( 140 )
CREATEINDEX BASETABLE DEFAULT \NEWYORK PRI ( 140 )

==  Set up default pools of scratch files for the sort
processes.
CREATEINDEX INDEX DEFAULT SCRATCH ($TEMP1,$TEMP2,$TEMP3)
CREATEINDEX INDEX DEFAULT \NEWYORK SCRATCH ($TEMP4,$TEMP5)

==  Request that overflow scratch files avoid certain
==  disks--those specified plus $SYSTEM and TM/MP audit
==  trail disks.
CREATEINDEX INDEX DEFAULT NOSCRATCHON ($SYS*,$WORK*)
==  Request that overflow scratch files use specific
==  disks on the remote node.

CREATEINDEX INDEX DEFAULT \NEWYORK SCRATCHON ($TEMP*)
==  Request that the $data3 sort process use $temp7 for
==  scratch space.
CREATEINDEX INDEX \NEWYORK.$data3 SCRATCH ($TEMP7)

== End of Configuration File
```

For detailed information about loading index partitions in parallel, including the syntax of the configuration file, see “Parallel Index Loading” in the *SQL/MP Reference Manual* or use the SQLCI HELP command.

Creating Constraints on Data

The SQL/MP data dictionary provides for data validity checking. Application programs do not have to perform data verification and checking, because a constraint defined on a table ensures that SQL performs the checking. Constraints provide for independence between data and code.

With SQL/MP, the definition of a constraint specifies a rule that all rows in the table must satisfy. The RDBMS enforces the constraint criteria when the constraint is created on a table with existing data and when rows of the table are updated or inserted. You can drop or add constraints at any time, as validity requirements for the data change, without affecting the application programs.

Constraints created on a table ensure that any data entered into the table satisfies the rules imposed by the constraints. To create a constraint on a table, use the CREATE CONSTRAINT statement.

For additional information on constraints and related performance issues, see Checking Data Integrity on page 14-23.

Using the CREATE CONSTRAINT Statement

The CREATE CONSTRAINT statement enforces these rules:

- Constraint names are SQL identifiers that can contain at most 30 of these characters: letters (A-Z, a-z), digits (0-9), and the underscore (_). The name must begin with a letter. SQL/MP reserved words, listed in the *SQL/MP Reference Manual*, are not allowed.

  These are examples of constraint names:

  ```
  VALID_EMPLOYEE_NUMBER
  VALID_JOB_CODES
  VALIDENDDATE
  MAXIMUM_SALARY
  ```

- Although you can specify constraint names in the CREATE CONSTRAINT statement in either uppercase or lowercase letters, the internal format is always the same: uppercase letters. So, the constraint names `MAXIMUM_SALARY` and `maximum_salary` are equivalent.
- The SYSKEY column is not allowed in the search condition defining a constraint.
- The CREATE CONSTRAINT statement requires an exclusive open of the underlying table, including all partitions, to ensure that no rows are inserted during the creation of the constraint. To add a constraint on a table loaded with data, the system verifies that all rows in the table satisfy the constraint. On a very large table, this processing can run for an extended time. You should create a constraint when the application is not active.
- The CREATE CONSTRAINT statement requires an exclusive table.
- You cannot create constraints directly on views. The constraints on underlying tables, however, affect the dependent views.

Additional Guidelines

When defining constraints, also consider these guidelines:

- You can use collations in the search condition defining a constraint.
- Aggregate functions and subqueries are not allowed in the search condition defining a constraint.
- For any given row of a table, the constraint must be able to be resolved by checking only that row.
- Constraint names should be as descriptive as possible.
- If you create a comment on a constraint, applications can use the comment text for routines that handle errors related to that constraint.
- Ensure that constraints on the same table are not logically in conflict. A conflict could cause all rows to be invalid.

  The CONSTRNT catalog table contains the description of constraints for tables recorded in the catalog. You can query this table to display the constraints on a table.

  This example queries the CONSTRNT table but first sets VARCHAR_WIDTH to 255 so that 255 characters of each row of the constraint definitions are displayed instead of 80 characters (the default width):

  ```
  >> SET STYLE VARCHAR_WIDTH 255;
  >> SELECT * FROM CONSTRNT
  +>     WHERE TABLENAME = "\SYS1.$VOL1.PERSNL.EMPLOYEE";
  ```
- To place a constraint on a particular partition, include the partition keys as part of the WHERE clause criteria in the search condition that defines the constraint.
- Whenever possible, you should create constraints after creating the table but before loading data into the table. If you create the constraint before loading the table, the data integrity of the table is ensured, because the LOAD utility does not put rows into a table if the rows do not conform to constraints. Depending on the error limit specified in the ALLOWERRORS option, the load operation either fails when encountering a row that does not conform to the constraints or loads only rows that do conform.
- If you are creating a constraint on an existing table, you should perform an interactive query on the table that is a negation of the constraint. This query identifies rows that violate the constraint. You should change or delete any identified rows before creating the constraint. The CREATE CONSTRAINT operation fails if rows in the table do not satisfy the constraint.
- You should not create a constraint on a loaded table within a user-defined TMF transaction because the transaction could overflow the TMF audit trails and cause an error. The CREATE CONSTRAINT statement automatically initiates several TMF transactions, as necessary, but performs tests outside a TMF transaction to

ensure that the rows satisfy the constraint. With this testing technique and the automatic transactions, the operation minimizes the TMF overhead of a potentially very long transaction and reduces output to the audit trails.

Examples of Creating Constraints

This example creates a constraint that checks for a valid employee number in the range 0 through 9999. The INFO DEFINE command displays DEFINEs used to identify the table for the constraint.

```
-- DEFINEs were previously added during this SQLCI
-- session or inherited from the command interpreter.
>> INFO DEFINE =EMPLOYEE;
     DEFINE NAME            =EMPLOYEE
     CLASS                  MAP
     FILE                   \SYS1.$VOL1.PERSNL.EMPLOYEE
>>  CREATE CONSTRAINT VALID_EMPLOYEE_NUMBER
+>      ON =EMPLOYEE
+>      CHECK EMPNUM BETWEEN 0 AND 9999;
```

This example creates a constraint, VALID_JOB_CODES, that ensures job code values are between 100 and 900, and the department is greater than or equal to 2500 but less than 5000. This constraint could apply to a partition of the table.

```
>>  CREATE CONSTRAINT VALID_JOB_CODES
+>      ON \SYS1.$VOL1.PERSNL.DEPT
+>      CHECK JOBCODE BETWEEN 100 AND 900
+>        AND (DEPTNUM >= 2500 AND
+>             DEPTNUM <  5000);
```

This example creates a constraint, VALID_PRICE_RANGE, that checks for a valid markup range in which the price must be greater than or equal to a 20 percent markup but less than or equal to a 200 percent markup:

```
>>  CREATE CONSTRAINT VALID_PRICE_RANGE
+>      ON $VOL3.INV.PARTLIST
+>      CHECK PRICE >= PARTS_COST * 1.20 AND
+>            PRICE <= PARTS_COST * 2.00;
```

This example creates a constraint, MIN_INVENTORY_FACTOR, that ensures that the quantity ordered is less than or equal to the quantity on hand or that the quantity ordered can be available by the estimated date when additional parts will be available:

```
>>  CREATE CONSTRAINT MIN_INVENTORY_FACTOR
+>      ON $VOL3.INV.INVFILE
+>      CHECK QTY_ORDERED <=  QTY_ON_HAND  OR
+>            (QTY_ORDERED <=  QTY_ON_HAND + QTY_RECEIVED AND
+>              DELIV_DATE > EST_RECEIVED_DATE);
```

This example creates a constraint to ensure that the value of the EMPNUM column is greater than 0 and less than 9999:

```
>>  CREATE CONSTRAINT VALID_EMPLOYEE_NUMBER
+>      ON $VOL1.PERSNL.EMPLOYEE
+>      CHECK EMPNUM > 0 AND
+>            EMPNUM < 9999;
--- SQL operation complete.
```

For the constraint VALID_EMPLOYEE_NUMBER, you might create this comment:

```
>> COMMENT ON CONSTRAINT VALID_EMPLOYEE_NUMBER
+>  ON $VOL1.PERSNL.EMPLOYEE
+>     IS "VALID EMPLOYEE NUMBERS ARE 1 to 9998";
--- SQL operation complete.
```

This example creates a constraint to ensure that the delivery date is greater than or equal to the order date:

```
>>  CREATE CONSTRAINT VALID_DELIV_DATE
+>      ON  $VOL.SALES.ORDERS
+>      CHECK DELIV_DATE >= ORDER_DATE;
--- SQL operation complete.
```

Creating Collations

A collation is an SQL object that contains rules for:

- Collating sequence (the sequence in which characters are ordered for sorting)
- Case (upshifting and downshifting)
- Character class

Collations can be applied to single-byte character columns in SQL tables.

You can define and create a collation and then associate the collation with a column in an SQL table. For example, you can define a collation by using the CREATE COLLATION statement that sorts characters in a different order than their character codes dictate. (If you do not specify a collation for a column, or if you specify the COLLATE CHARACTER SET clause, SQL collates the column according to the binary values of the data.)

Then when you create a column that has a character data type and a single-byte character set by using a CREATE TABLE or ALTER TABLE statement, you can specify the name of the collation to associate with the column. If the column is part of the primary key for the table, the collation also affects the storage order for rows in the table.

A collation name must be a Guardian name.

SQL allows a collation (or a class MAP DEFINE name that points to a collation) in the COLLATE clause of these statements:

- CREATE TABLE, ALTER TABLE...ADD COLUMN, or CREATE VIEW statement to specify a default collating sequence for one or more columns in a table or view
- The GROUP BY or ORDER BY clause in a SELECT statement to override the default collating sequence for a column

- An SQL expression in a SELECT statement to specify different collating sequences (including upshifting) for character strings
- The WHERE clause in an UPDATE or DELETE statement to specify a different collating sequence for a column
- The CREATE INDEX statement to specify an order that differs from the base table
- The CREATE CONSTRAINT statement to specify a collating sequence for a column in the constraint

For the syntax of these statements, see the *SQL/MP Reference Manual*.

Creating Collation Source Files

To create a collation source file, you enter rules for character processing into an EDIT file in collation compiler syntax. The *SQL/MP Reference Manual* describes the syntax for specifying these rules in a collation source file. You must create a collation source file for each collation specified in your database design.

You then create the collation by executing the CREATE COLLATION statement either from SQLCI or embedded in a C program. The CREATE COLLATION statement calls the collation compiler to compile the collation source file and generate the collation. This statement also registers the collation in the SQL catalog.

After you create a collation, you can use the ALTER COLLATION statement to rename it or change its security string or owner if necessary. To delete a collation, you can use the DROP COLLATION statement, as long as the collation is not used by any SQL objects or programs. HP also provides a set of system procedures you can use to manipulate collations (for example, to read a collation or to compare two collations).

Example 5-4 shows a sample source file.

Example 5-4. Collation Source File (page 1 of 3)

```
LC_COLLATE

#   This file contains character processing rules.
#
#   The collating sequence sorts most of the accented forms of a, e, i,
#   o, and u equal to the unaccented form. The collating sequence is
#   case insensitive.
#
#   Upshift for a, e, i, o, u -grave -acute -circumflex is
#   A, E, I, O, U. Upshift for e-umlaut is E. Upshift for i-umlaut
#   is I, and upshift for y-acute is Y.
#
#   These character processing rules are for the ISO88591 character set.

#   Start the collating orders.
order_start      forward
     \d032      \d032      #  32 = space  #
     \d160      \d032      #  NBSP        #
     <0>        <0>
     ...       ...
```

Example 5-4. Collation Source File (page 2 of 3)

```
     <9>        <9>
     <A>        <A>
     <Z>        <Z>
     <a>        <A>
     ...       ...
     <z>        <Z>
     \d192      <A>        #  192 - 195, 224 - 227 are various forms
     ...       <A>        #  of “A” and “a”
     \d195      <A>
     \d224      <A>
     ...
     \d227      <A>
     \d199      <C>        #  199 = C-cedilla
     \d231      <C>        #  231 = c-cedilla
     \d208      <D>        #  208 = Eth
     \d240      <D>        #  240 = eth
     \d200      <E>        #  200 - 203, 232 - 235 are various forms
     ...       <E>        #  of “E” and “e”
     \d203      <E>
     \d232      <E>
     ...       <E>
     \d235      <E>
     \d204      <I>        #  204 - 207, 236 - 239 are various forms
     ...       <I>        #  of “I” and “i”
     \d207      <I>
     \d236      <I>
     ...       <I>
     \d239      <I>
     \d209      <N>        #  209 = N-tilde
     \d241      <N>        #  241 = n-tilde
     \d210      <O>        #  210 - 213, 242 - 245 are various forms
     ...       <O>        #  of “O” and “o”
     \d213      <O>
     \d242      <O>
     ...       <O>
     \d245      <O>
     \d217      <U>        #  217 - 219, 249 - 251 are various forms
     ...       <U>        #  of “U” and “u”
     \d219      <U>
     \d249      <U>
     \d224      <U>
     ...       <U>
     \d221      <Y>        #  221 = Y-acute
     \d253      <Y>        #  253 = y-acute
     \d255      <Y>        #  255 = y-acute
     \d198      \d198      #  198 = AE
     \d230      \d230      #  230 = ae
     \d216      \d216      #  216 = O-slash
     \d248      \d248      #  248 = o-slash
     \d197      \d197      #  197 = A-ring
     \d229      \d197      #  229 = a-ring
     \d222      \d222      #  222 = Thorn
     \d254      \d222      #  254 = thorn
     \d033      <!>        #  33 - 47 are symbols encoded in
     ...       ...       #  sequences
     \d047      </>
     \d173      <->        #  173 = SHY
     \d058      <:>        #  58 - 63 are symbols encoded in
     ...       ...       #  sequences
     \d063      <?>
     \d064      <@>
     \d091      <[>        #  91 - 96 are symbols encoded in
     ...       ...       #  sequences
```

Example 5-4. Collation Source File (page 3 of 3)

```
     \d096      <`>
     \d123      <{>       #  123 - 126 are symbols encoded in
     ...       ...       #  sequences
     \d126      <~>
     \d127      IGNORE

     \d196      "<a><e>"   #  A-umlaut is sorted as a string of a and e
     \d228      "<a><e>"   #  a-umlaut is sorted as a string of a and e
     \d214      "<o><e>"   #  O-umlaut is sorted as a string of o and e
     \d246      "<o><e>"   #  o-umlaut is sorted as a string of o and e
     \d220      "<u><e>"   #  U-umlaut is sorted as a string of u and e
     \d252      "<u><e>"   #  u-umlaut is sorted as a string of u and e
     \d223      "<s><s>"   #  sharp-s is sorted as a string of s and s
     \d163      \d163      #  upper half specials and controls
     \d215      \d215      #  multiply sign

.
.
.
     UNDEFINED IGNORE
order_end

END LC_COLLATE

LC_CTYPE

charclass  alphas; numerics; specials
alphas     <A>;...;<Z>;\
           <a>;...;<z>;\
           \d192;...;\d214;\d216;...;\d246;\d248;...;\d255

numerics   <0>;<1>;\d050;\x33;\
           <4>;...;<9>

specials

toupper    (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);(<f>,<F>);\
           (<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);(<k>,<K>);(<l>,<L>);\
           (<m>,<M>);(<n>,<N>);(<o>,<O>);(<p>,<P>);(<q>,<Q>);(<r>,<R>);\
           (<s>,<S>);(<t>,<T>);(<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);\
           (<y>,<Y>);(<z>,<Z>);\
           (\d224,\d065);(\d225,\d065);\
           (\d226,\d065);(\d227,\d195);\
           (\d231,\d199);\
           (\d236,\d073);(\d237,\d073);\
           (\d238,\d073);(\d239,\d073);\
           (\d241,\d209);\
           (\d242,\d079);(\d243,\d079);\
           (\d244,\d079);(\d245,\d213);\
           (\d249,\d085);(\d250,\d085);\
           (\d251,\d085);\
           (\d255,\d089);\
           (\d229,\d197);\
           (\d248,\d216);\
           (\d230,\d198);\
           (\d254,\d222);\
           (\d228,\d196);\
           (\d252,\d220)
END LC_LCTYPE

LC_TDMCODESET
ISO88591               #  This specifies the ISO88591 character set.
END LC_TDMCODESET
```

Creating Collation Objects

To create SQL collation objects, you must have the collation compiler compiled as an SQL program and registered in a catalog, as explained under Installing SQL/MP on page 2-2. When you issue the CREATE COLLATION statement, SQL invokes the collation compiler to compile the source file you specify.

Note. Consider the locations for your collations carefully. Moving a collation after you create dependent objects is difficult. To move a collation, you must stop transaction processing on all dependent objects, save the object definitions and data, delete the dependent objects, copy the collation to the new location, delete the original collation, and then re-create the dependent objects.

Create all the collations for your database first, before you create the objects that refer to collations. To create a collation, you can specify an existing collation source file in a CREATE COLLATION statement, as in this example:

```
>> CREATE COLLATION COLL1
>>   FROM =COLLATE1
>>   CATALOG =SALES ;
```

The DEFINE =COLLATE1 specifies the source file.

You can also create a collation like another collation, without specifying a source file:

```
>> CREATE COLLATION COLL2
>>   LIKE COLL1
>>   CATALOG =INVENT ;
```

Each collation has a single-byte character set associated with it, stored in the CHARACTERSET column of the CPRULES catalog table. This character set must match the character set associated with a column when you use the COLLATE clause to define or specify an index for a column.

Securing Collations

When you define a collation, SQL assigns it your current default security. You can alter this security, but be careful. If you alter the security so that fewer users have access to the collation, you also restrict the access to any table, index, view, or program that uses the collation.

For example, suppose that the database administrator creates a collation and makes it available to an application developer, and the developer creates a table that uses the collation. Then the DBA alters the security to a string that takes access to the collation away from the developer. When the developer attempts to compile SQL statements that refer to the table, the compilations fail because of security violations. SQL reports the name of the collation causing the violations, however.

To alter the security of a collation, you must have authority to read and write to the collation and the catalog in which the collation is registered. You can alter a collation's security by using the ALTER COLLATION statement or the SECURE command.

6 Querying SQL/MP Catalogs

SQL/MP catalogs contain information you can use to manage the database. Obtaining information from the catalogs with SQL queries can help you determine the current status of the database. You can also use the FILEINFO and VERIFY utilities to show the status of objects and the DISPLAY USE OF utility to show usage relationships.

Determining Object and Program Dependencies

The USAGES catalog table stores information about dependencies between objects. A dependent object and the object on which it depends can be registered in different catalogs; therefore, the USAGES tables in both catalogs have an entry about the dependency. This table lists initial and dependent objects:

Initial Object	**Dependent Objects**
Table	Index Protection view Shorthand view SQL object program
Protection view	Shorthand view SQL object program
Shorthand view	Shorthand view SQL object program
Index	SQL object program
Collation	Table Index Protection view Shorthand view SQL object program

Note. If you specify the NOREGISTER option for a program, as described under Moving Programs on page 10-39, SQL/MP does not register the program in the catalog and it will not appear as a dependent object. In addition, the program can be installed at a different location (subvolume or node) without recompilation or registration.

The next example illustrates corresponding dependency entries for a table and a dependent program whose definitions are stored in different catalogs on different nodes.

```
\SYS1.$V.CAT1 Catalog                  \SYS2.$V.CAT2 Catalog

USAGES Table                           USAGES Table

\SYS1.$V.SV.TABLEA has dependent       \SYS2.$V.P.PROG2 uses
   object \SYS2.$V.P.PROG2                \SYS1.$V.SV.TABLEA
```

You can display dependent objects by using the DISPLAY USE OF utility or by directly querying individual USAGES tables. The DISPLAY USE OF utility is convenient to use because it searches multiple USAGES tables for dependencies. The system prepares the display by searching all the catalogs for all the objects that depend directly or indirectly on the object you specify in the command.

Using the DISPLAY USE OF Command

The DISPLAY USE OF command enables you to specify a table, index, view, or collation. The utility displays the objects that depend on the specified object.

The DISPLAY USE OF utility has two formats: standard and brief. The standard format is the default. The brief format does not display the partition flags, the owners, the security of the objects, or the catalog name.

The next examples show the standard and brief formats of the DISPLAY USE OF utility. Each example includes the DISPLAY USE OF command and the information displayed.

This example shows the standard format:

```
>> DISPLAY USE OF INVENT.SUPPLIER;

   Object Name                           Type S P  Owner Name       Secure
   ------------------------------------- ---- - -- ---------------- ------
          Catalog Name
          --------------------------------

 0 \SYS1.$VOL.INVENT.SUPPLIER             TA        AUSER   .ERIC     NCNC
        $SQL.INVENT
   1 \SYS1.$VOL.INVENT.XSUPPNAM           IN        AUSER   .ERIC     NCNC
          $SQL.INVENT
     2 \SYS1.$VOL.AUSERSV.OPROG2          PG        AUSER   .PAT      NCNC
            $SQL.SALES
   1 \SYS1.$VOL.AUSERSV.OPROG2            PG  *

U = Undefined node          N = Node unavailable       T = Unsupported type
@ = Node not in list        * = Previously displayed   ? = System error

Number of unique dependencies :  2
Number of direct dependencies :  1
```

This example shows the brief format:

```
   Object Name                             Type S
   --------------------------------------- ---- -
0  \SYS1.$VOL.INVENT.SUPPLIER               TA
   1 \SYS1.$VOL.INVENT.XSUPPNAM             IN
     2 \SYS1.$VOL.AUSERSV.OPROG2            PG
   1 \SYS1.$VOL.AUSERSV.OPROG2              PG   *

U = Undefined node           N = Node unavailable       T = Unsupported type
@ = Node not in list         * = Previously displayed   ? = System error

Number of unique dependencies :  2
Number of direct dependencies :  1
```

The DISPLAY USE OF utility does not accept OSS path names as parameters, but does display OSS programs if they are dependent on the list of requested objects. The name of an SQL program stored in an OSS file is displayed as its Guardian file name equivalent and then in its path name format. If there is more than one path name linked to the program, only one path name is displayed (the first path name available to the current user). If the OSS path name is not accessible to the user, SQL returns “No path name is accessible” instead of the path name.

The code for an OSS object is PG (same as for SQL programs stored in Guardian files).

Displaying Information About Usages by Querying the Catalog

This example displays all dependency information stored in the specified catalog. The VOLUME command indicates the default volume and designates the subvolume name, which is also the catalog name, as the default subvolume. This SELECT statement does not need to include the fully qualified name of the catalog table:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT *  FROM USAGES;
```

Note: If you query the catalog tables directly, the results of your query can be affected by the version level of the catalog, as described under Displaying Catalog, Object, and Program Versions on page 6-15.

Displaying Current Database Definitions

You can query the catalog tables to display current database definitions. When selecting information from the catalog tables, you must either use fully qualified names or use partially qualified names in conjunction with the LIKE predicate. You must use uppercase letters for data, but you can use either uppercase or lowercase letters for column names. Instead of typing all uppercase letters, you can use the UPSHIFT function or a COLLATE clause that specifies a case-insensitive collation.

For the descriptions of the columns of the catalog tables, see the *SQL/MP Reference Manual*.

Displaying Information About Catalogs

Information about all the catalogs on the node appears in the system directory of catalogs, which is the CATALOGS table in the system catalog.

The CATALOGS table must reside on the subvolume SQL on the same volume as the system catalog. The default location is $SYSTEM.SQL.

This query displays all SQL catalogs on the node \SYS1:

```
>>  SELECT * FROM \SYS1.$SYSTEM.SQL.CATALOGS;
```

Displaying Information About Tables

Catalogs have three tables that describe base tables:

- BASETABS describes the attributes of a base table, such as whether any constraints are defined on the table and the number of rows in the table.
- TABLES describes file-label information, such as the security string, owner ID, creation timestamp, and number of columns.
- FILES describes file-label information, such as the file type (organization), extents, audit flag, partitioned flag, address of the end of file, record size, and various other flags.

You can query each catalog table separately, or you can create a joined view of the tables.

This example displays BASETABS information about a base table:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM BASETABS
+>      WHERE TABLENAME = "\SYS1.$VOL1.SALES.CUSTOMER";
```

This example displays TABLES information about a table:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM TABLES
+>      WHERE TABLENAME = "\SYS1.$VOL1.SALES.CUSTOMER";
```

This example displays FILES information about a table. The query uses the LIKE predicate with the wild-card character % to indicate that a string ranging from no characters to many characters is acceptable in the wild-card position for a qualifying file name. Also, notice that characters in LIKE predicates are case sensitive and that references to data contained in tables are in uppercase letters.

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM FILES
+>      WHERE FILENAME LIKE "%CUSTOMER%";
```

This example displays tables and views described in TABLES for a given owner ID. For this example, the group ID is 240, and the user ID is 100.

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT TABLENAME, TABLETYPE
+>      FROM TABLES
+>      WHERE GROUPID = 240 AND USERID = 100;
```

This example displays selected columns from a join of the TABLES and FILES catalog tables. The search condition for the join operation is WHERE TABLENAME = FILENAME.

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT TABLENAME, TABLETYPE, GROUPID, USERID,
+>         SECURITYVECTOR, AUDIT, FILETYPE, EOF
+>       FROM TABLES, FILES
+>       WHERE TABLENAME = FILENAME;
```

Displaying Information About Views

Catalogs have two tables that describe views:

- VIEWS describes the attributes of a view, such as the text of the view definition and whether the view is a protection or shorthand view.
- TABLES describes some file-label information, such as the security string, owner ID, creation timestamp, and number of columns.

You can query each catalog table separately, or you can create a joined view of the tables.

The default width for displaying view text (with varying-length data) is 80 characters. For example, you can use the SET STYLE command to specify a VARCHAR display width of 255 characters. This width ensures that you can display 255 characters of the view text.

This example displays VIEWS information for a view:

```
>> VOLUME \SYS1.$VOL1.PERSNL;
>> SET STYLE VARCHAR_WIDTH 255;
>> SELECT * FROM VIEWS
+>     WHERE VIEWNAME = "\SYS1.$VOL1.PERSNL.EMPLIST";
```

This example displays TABLES information for a view:

```
>>  VOLUME \SYS1.$VOL1.PERSNL;
>>  SELECT * FROM TABLES
+>      WHERE TABLENAME = "\SYS1.$VOL1.PERSNL.EMPLIST";
```

This example displays view names and the audit flag settings of all views. The VALIDDEF column of the display shows Y or N to indicate the validity of a view.

```
>>  VOLUME \SYS1.$VOL1.PERSNL;
>>  SELECT VIEWNAME, VALIDDEF, AUDIT
+>      FROM VIEWS;
```

This example displays selected columns from a join of the TABLES and VIEWS catalog tables for group ID 240:

```
>>  VOLUME \SYS1.$VOL1.PERSNL;
>>  SELECT VIEWNAME, GROUPID, USERID, SECURITYVECTOR,
+>>         AUDIT, PROTECTION, WITHCHECKOPTION
+>      FROM TABLES, VIEWS
+>      WHERE VIEWNAME = TABLENAME AND
+>            GROUPID  = 240;
```

Displaying Information About Constraints

Information about constraints on a table appears in the CONSTRNT catalog table in the catalog that contains the table description.

To determine whether a table has constraints, you can query the BASETABS table to check whether the constraints flag is set. If CONSTRAINTS is Y, one or more constraints exist. If CONSTRAINTS is N, no constraint exists.

This example displays the TABLENAME and CONSTRAINTS columns:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT TABLENAME, CONSTRAINTS
+>      FROM BASETABS
+>      WHERE TABLENAME = "\SYS1.$VOL1.SALES.ORDERS";
```

The default width for displaying a constraint definition (with varying-length data) is 80 characters. For example, you can use the SET STYLE command to specify a VARCHAR display width of 255 characters.

This example displays all the current constraints on a table:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SET STYLE VARCHAR_WIDTH 255;
>>  SELECT * FROM CONSTRNT
+>      WHERE TABLENAME = "\SYS1.$VOL1.SALES.ORDERS";
```

Displaying Information About Collations

Catalogs have two tables that describe collations:

- CPRULES describes characteristics of collations, including whether the collation ever considers two different characters as being equal (for example, whether ‘A’ = ‘a’), the character set to which the collation rules apply, and the size and version of the collation.
- CPRLSRCE describes the collation source definition, including the entire source text that specifies the character processing rules.

This example displays CPRULES information about a collation:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM CPRULES
+>      WHERE CPRULESNAME = "\SYS1.$VOL1.SALES.FRENCH";
```

This example displays CPRLSRCE information about a collation:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM CPRLSRCE
+>      WHERE CPRULESNAME LIKE "%FRENCH%";
```

Displaying Information About Columns

Information about columns appears in the COLUMNS catalog table and includes the data type definitions.

You can query the COLUMNS table for information about particular column definitions, or you can obtain a list of columns for a specific table. You might also query the COLUMNS table to check the definitions of columns whose definitions must match.

This example displays all column names for a table and the data type definitions:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT COLNAME, DATATYPE, COLSIZE, SCALE
+>      FROM COLUMNS
+>      WHERE TABLENAME = "\SYS1.$VOL1.SALES.ORDERS";
```

This example displays all tables in which the column PRICE is found and includes the data definitions of the columns:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT COLNAME, TABLENAME, DATATYPE, COLSIZE, SCALE,
+>      PRECISION, PICTURETEXT, CHARACTERSET
+>      FROM COLUMNS
+>      WHERE COLNAME = "PRICE";
```

Displaying Comments and Help Text

Comments specified for an object are located in the COMMENTS catalog table. You can query this table to display the comments.

This example selects all the comments on the table CUSTOMER:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM COMMENTS
+>      WHERE OBJNAME = "\SYS1.$VOL1.SALES.CUSTOMER";
```

This example selects all the comments on all the constraints defined for the table CUSTOMER. Constraints are denoted by the value “CN” in the OBJTYPE column of the COMMENTS table.

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM COMMENTS
+>      WHERE OBJNAME = "\SYS1.$VOL1.SALES.CUSTOMER" AND
+>          OBJTYPE = "CN";
```

The COMMENTS catalog table also records help text for columns. You can query this table to display the help text. The OBJSUBNAM column of the COMMENTS table contains column names, and the OBJTYPE column contains “HC” to denote help text.

This example selects the help text for the column LAST_NAME from the table EMPLOYEE:

```
>>  VOLUME \SYS1.$VOL1.PERSNL;
>>  SELECT * FROM COMMENTS
+>      WHERE OBJNAME = "\SYS1.VOL1.PERSNL.EMPLOYEE" AND
+>          OBJSUBNAME = "LAST_NAME" AND
+>          OBJTYPE = "HC";
```

Displaying Information About Indexes

Three catalog tables contain information about keys and indexes:

- KEYS describes the columns of each primary key and index.
- INDEXES describes index-file information and includes an entry for the primary key of each base table.
- FILES describes file-label information for each index, such as the file type, number of extents, audit flag, partitioned flag, end-of-file address, and record size.

This example displays values from the TABLENAME column of the CUSTOMER table:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT TABLENAME
+>      FROM BASETABS
+>      WHERE TABLENAME = "\SYS1.$VOL1.SALES.CUSTOMER";
```

This example displays all indexes for the table CUSTOMER that are described in the catalog \SYS1.$VOL1.SALES:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM INDEXES
+>      WHERE TABLENAME = "\SYS1.$VOL1.SALES.CUSTOMER";
```

This example displays key information for the table CUSTOMER in a joined view of the INDEXES and KEYS catalog tables:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT K.INDEXNAME, KEYSEQNUMBER, TABLECOLNUMBER, ORDERING
+>      FROM KEYS K, INDEXES I
+>      WHERE TABLENAME = "\SYS1.$VOL1.SALES.CUSTOMER" AND
+>          K.INDEXNAME = I.INDEXNAME;
```

This TACL macro displays the keys and indexes of a table:

```
?TACL MACRO
#FRAME
#PUSH #INLINEECHO, #INLINEPREFIX
#SET #INLINEECHO -1
#SET #INLINEPREFIX //
#PUSH tname
#SET tname [#SHIFTSTRING /UP/ %1%]
$SYSTEM.SYSTEM.SQLCI /INLINE/
// VOLUME %2%;
//
// -- Report on the different keys and indexes of a table:
//
// SELECT I.TABLENAME, KEYTAG, KEYSEQNUMBER, COLNAME, ORDERING
//   FROM COLUMNS C, KEYS K, INDEXES I
//     WHERE C.TABLENAME = I.TABLENAME AND
//           I.INDEXNAME = K.INDEXNAME AND
//           I.TABLENAME LIKE "%%[tname]%%" AND
//           K.TABLECOLNUMBER = C.COLNUMBER
//     ORDER BY I.TABLENAME, KEYTAG, KEYSEQNUMBER
//     ;
// EXIT;
#UNFRAME
```

For more information about TACL macros and how to write them, see the *TACL Reference Manual*.

This example displays attribute information for the index XORDCUS. The query uses the LIKE predicate with the wild-card character % to indicate that a string of 0, 1, or more characters is acceptable in each wild-card position for a qualifying file name.

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT * FROM FILES
+>      WHERE FILENAME LIKE "%XORDCUS%";
```

Displaying Information About Partitions

Information about partitions appears in the PARTNS catalog table. Your first query about partitions should determine whether the object has partitions.

To obtain information on partitions, query the FILES table to determine whether the partitions flag is set for the object. If PARTITIONED is Y, the object has partitions; if PARTITIONED is N, the object is not partitioned.

This example displays the PARTITIONED column value for the PARTLOC table:

```
>>  VOLUME \SYS1.$VOL1.INVENT;
>>  SELECT FILENAME, PARTITIONED
+>      FROM FILES
+>      WHERE FILENAME = "\SYS1.$VOL1.INVENT.PARTLOC";
```

This example displays all the partition information for the PARTLOC table. The query uses the LIKE predicate with the wild-card character % to indicate that a string of 0, 1, or more characters is acceptable in the wild-card position for a qualifying name.

```
>>  VOLUME \SYS1.$VOL1.INVENT;
>>  SELECT * FROM PARTNS
+>      WHERE FILENAME LIKE "%PARTLOC%";
```

Joining Catalog Tables With UNION

The UNION operator in a SELECT statement can effectively join catalog tables of the same type.

Suppose that you want to join the TABLES catalog tables from two catalogs, SALES and INVENT, to produce a joined output display. These catalogs, in fact, reside on two different nodes. This SELECT statement selects the TABLENAME, TABLECODE, and SECURITYVECTOR columns but eliminates tables in which TABLECODE is greater than or equal to 572:

```
>>  VOLUME \SYS1.$VOL1.SALES;
>>  SELECT TABLENAME, TABLECODE, SECURITYVECTOR
+>          FROM \SYS1.$DATA2.SALES.TABLES
+>          WHERE TABLECODE < 572
+>      UNION SELECT TABLENAME, TABLECODE, SECURITYVECTOR
+>          FROM \SYS2.$DATA7.INVENT.TABLES
+>          WHERE TABLECODE < 572
+>      ORDER BY TABLECODE;
```

Displaying File and Security Attributes

The FILEINFO utility displays file and security attributes for catalog tables, base tables, indexes, views, collations, and Enscribe and OSS files. You can request a display for one object or for each object specified in a qualified file set list.

You can request that the file information be displayed in brief format (a condensed, one-line format) or in the detailed, standard format. You can also request statistical information about the data bit map.

This command obtains a brief format listing for all objects described in the $VOL1.INVENT catalog:

```
>>  FILEINFO *.*.* FROM CATALOG $VOL1.INVENT;
```

This command obtains a brief format listing from the $VOL1.INVENT catalog for SQL tables that have not been updated since January 10, 1989:

```
>>  FILEINFO *.*.* FROM CATALOG $VOL1.INVENT
+>   WHERE SQL AND MODTIME BEFORE JAN 10 1989;
```

If your files are managed by SMF, you typically create an SQL object with a logical file name. You can find out the corresponding physical file name by using the detailed format of the FILEINFO command and specifying the logical name.

This command obtains a detailed format listing, including the physical file name, for the SQL table whose logical name is $VIR1.SALES.ORDERS (only part of the listing is shown):

```
>>  FILEINFO $VIR1.SALES.ORDERS, DETAIL;

$VIR1.SALES.ORDERS                     20 Aug 1996, 11:41
    SQL BASE TABLE
    CATALOG $VIR1.SALES
    |
    |
    PHYSICAL NAME: $PVOL11.ZYS00025.A045OR02
```

If you know a physical file name and want to find out the corresponding logical name, you can use the FILEINFO command. This command displays the logical name corresponding to a specified physical file name:

```
>>  FILEINFO $PVOL11.ZYS00025.A045OR02, DETAIL;

$PVOL11.ZYS00025.A045OR02               20 Aug 1996, 11:41
    SQL BASE TABLE
    CATALOG $VIR1.SALES
    |
    |
    LOGICAL NAME: $VIR1.SALES.ORDERS
```

Suppose that your starting point is a physical volume. You can use the detailed format of the FILEINFO command to display the names of all the physical files managed by SMF that reside on a specified physical volume. In addition, the FILEINFO command displays the logical file names associated with those physical files. To perform this task, you must include the ZYS prefix in the subvolume part of the qualified file set. SMF reserves the ZYS prefix for physical file names.

This command displays the physical file names managed by SMF and residing on the volume $PVOL11. It also displays the logical file names (in addition to other file and security attributes) associated with those files.

```
>>  FILEINFO $PVOL11.ZYS*.*, DETAIL;
```

This command omits the ZYS prefix. This example does not display any physical files managed by SMF. It does display files residing on $PVOL11 that are outside the control of SMF.

```
>>  FILEINFO $PVOL11.*.*, DETAIL;
```

For OSS files, these considerations apply:

- You cannot specify an OSS path name as input to the FILEINFO command, but you can specify the Guardian ZYQ name associated with the OSS program.
- The owner and security are displayed as appropriate OSS values.
- The STATISTICS option is equivalent to the DETAIL option.

- The name of a program stored in an OSS file is displayed as its Guardian file name equivalent and then in its path name format. If there is more than one path name linked to the program, only one path name is displayed (the first path name available to the current user).
- Several informational items are not displayed because they do not apply to OSS files. For example, the EXTENTS option displays a message that EXTENTS information does not apply to an OSS file.

As an alternative, run FUP or OSS utilities to obtain information about an OSS file.

Determining Object Integrity and Consistency

Two tools are available to verify the definitional integrity of an object or to check the consistency of an object's internal blocks:

- The SQLCI VERIFY utility checks the definitional integrity of an object.
- The Guardian FILCHECK utility reports on the internal, physical data structure of objects and checks that the structure is consistent.

Using VERIFY to Check Definitional Integrity

The VERIFY utility determines the definitional integrity of an object. The utility does not verify data integrity. An object has definitional integrity if it is consistently described in all file labels and catalog tables and if the descriptions of all related objects are valid.

The database can become inconsistent through database changes that are not applied consistently throughout the related objects. SQL can prevent many of the operations that can cause inconsistency, but it does not always detect all the operational errors. For instance, when you drop a table, SQL attempts to drop all dependencies; however, a user could restore objects that might not be consistent with the related objects. System operational problems or system failures could also cause inconsistencies in the data dictionary.

You can use VERIFY to check catalogs, tables, views, indexes, collations, and programs. VERIFY checks each object for dependent relations or underlying table consistency, as required by the object.

For SQL programs stored in OSS files, VERIFY checks for consistency between the label and the catalog information for [IN]VALID, PFV, and PCV.

For information on invalid programs, see Determining Validity of a Program on page 10-4.

The VERIFY utility uses a qualified file set list to assist you in identifying a specific object or group of objects. You can verify programs and have the list of invalid programs written to an EDIT file. This example verifies all objects that are in the catalog SALES and that have a FILECODE value less than 572. This report does not verify catalog tables because their file codes are not less than 572.

```
>>  VERIFY *.*.* FROM CATALOG SALES WHERE FILECODE < 572;

---    Verifying $VOL1.SALES.ATABLE1
---    $VOL1.SALES.ATABLE1 verified.

---    Verifying $VOL1.SALES.AVIEW1
---    $VOL1.SALES.AVIEW1 verified.

---    Verifying $VOL1.SALES.AINDEX1
---    $VOL1.SALES.AINDEX1 verified.

--- SQL operation complete.
```

This example verifies the definition of a single object, the table EMPLOYEE:

```
>>  VERIFY DEF OF $VOL1.PERSNL.EMPLOYEE;

---    Verifying $VOL1.PERSNL.EMPLOYEE
---    $VOL1.PERSNL.EMPLOYEE verified.

--- SQL operation complete.
```

This example shows the error generated by an invalid object in the catalog SALES:

```
>> VERIFY *.*.* FROM CATALOG $DATA1.SALES;

--- Verifying $DATA1.SALES.ODETAIL
--- $DATA1.SALES.ODETAIL verified.

--- Verifying $DATA1.SALES.ORDERS
--- $DATA1.SALES.ORDERS verified.

--- Verifying $DATA1.SALES.ORDREP
*** ERROR from SQL [-1233]: The catalog entry in the TABLES
table
*** for \SQLA.$DATA1.SALES.ORDREP indicates an invalid COLCOUNT:
0.
*** ERROR from SQL [-9881]: Unable to obtain catalog definition
for
***  SQL shorthand view object.

--- SQL operation complete.
```

Using FILCHECK to Check Structural Consistency

The FILCHECK utility checks the physical structure of a DP2 structured object and reports any errors. The internal checks include:

- Forward and backward pointers in blocks
- Relative sector number and checksum of every block
- Correct index levels
- Data block and index block linkage and length
- Block headers and rows in relative files
- Offset pointers and order

FILCHECK does not correct any errors in the physical structure. If errors are detected, you might want to use TMF file recovery to recover the object. Alternatively, contact your service provider for other possible recovery methods. If you contact the Global Customer Support Center (GCSC), have your FILCHECK output available and determine whether full TMF recovery is a possibility.

You can request that the file information be displayed in brief format (a condensed summary format) or in the detailed standard format.

This example shows a brief format listing for the EMPLOYEE table from the catalog $VOL1.PERSNL:

```
14>  FILCHECK $VOL1.PERSNL.EMPLOYEE, BRIEF ;
     @@@@ CHECKING $VOL.PERSNL.EMPLOYEE
          Time:  18:36:57 3/28/89
          This is an audited table
          SQL root table
          SQL integrity constraints present.
          Key-Sequenced file
          >Begin Summary Report.
           Data blocks checked   : 5
           Data records checked  : 190
           Index blocks checked  : 1
           Index records checked : 5
           Free block checked    : 0
           Bitmap blocks checked : 1
           Soft bitmap errors    : 0
          >End of summary report.
```

Displaying Catalog, Object, and Program Versions

The different product version updates (PVUs) of SQL/MP produce and support different versions of SQL catalogs, objects, and programs, as discussed in detail in the *SQL/MP Version Management Guide*. Because the behavior and capabilities of a catalog or object depend on its version, you must be able to determine the version of catalogs or objects that your applications or interactive queries reference. The catalog tables contain version information, which you can display, but you can also use a GET VERSION statement to retrieve this information for you as follows:

```
>> GET VERSION OF TABLE $VOL1.PERSNL.EMPLOYEE;
   VERSION:  310
   --- SQL operation complete.
```

To retrieve information about any of the three types of program versions (PCV, PFV, and HOSV), use the GET VERSION OF PROGRAM statement. When inquiring about an OSS file, specify the ZYQ name of the OSS file.

The VERSIONS catalog table provides version information about the catalog. You can use the GET VERSION statement to get this information for you as follows:

```
>> GET VERSION OF CATALOG $VOL1.$DATA1.PERSNL.VERSIONS;
   VERSION:  310
   --- SQL operation complete.
```

This information is also replicated for each catalog in the CATALOGS table in the system catalog. The catalog format version is located in the CATALOGFORMAT column, and the catalog version is located in the CATALOGVERSION column:

```
>> SELECT CATALOGFORMAT, CATALOGVERSION FROM catalog-
name.VERSIONS;
```

You can also obtain version information from the system catalog CATALOGS table, which contains the version of each catalog on the node, as shown:

```
>> SELECT CATALOGNAME, SUBSYSTEMNAME, VERSION
+>     FROM sys-cat-volume.SQL.CATALOGS
+>     WHERE CATALOGNAME = "\SYS1.$DATA1.PERSNL";

CATALOGNAME              SUBSYSTEMNAME      VERSION
-----------------------  -----------------  -------

\SYS1.$DATA1.PERSNL      SQL                     A310

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

In the information returned from either the VERSIONS or the CATALOGS table, the code A010 indicates a version 1 catalog, A011 indicates a version 2 catalog, A300 indicates a version 300 catalog, A310 indicates a version 310 catalog, A315 indicates a version 315 catalog, and so on.

You can also determine the version of SQL/MP tables, views, indexes, and collations by using the FILEINFO command with the DETAIL option. (You cannot use the FILEINFO command to obtain catalog versions.)

You can also obtain version information programmatically by issuing procedure calls. For more information about this method, see the *SQL/MP Version Management Guide*.

7 Adding, Altering, Removing, and Renaming Database Objects

After you create a database, you can assume that the database is consistent and that application data is valid. Database management operations must ensure the same level of data consistency and validity.

Any addition, alteration, or deletion to the database should be carefully planned. Only authorized persons should make changes to the active data dictionary.

You should review all changes to the database for these issues, discussed further in this section:

- Will dependent SQL objects be affected? Sometimes a single change makes other changes necessary for consistency. Your plan for completing a change should include performing an initial change and, if needed, changes to dependent SQL objects throughout the database.
- Does the user making the change have the authority to do so?
- Are the necessary base tables, partitions, and systems available?
- What impact would the change have on the production application? Should the application be stopped to apply this change consistently and without system degradation?
- Does the user making the change have a valid recovery mechanism to undo the change if required?

When you make changes to the database, you should always maintain a log of the operations of adding, dropping, and altering database objects.

Adding Objects to a Database

Because an SQL/MP database has an active data dictionary, you can add objects to the database online:

- Adding a catalog or table to the database is the same as creating the object for the first time. The addition can require integration into the existing database but does not affect the validity of the current database or its use.
- Adding a new dependency (an index, view, column, constraint, comment, or partition; or object, column or constraint that uses a collation) alters the current state of the database.

This table summarizes the objects that can be added to the database and the SQL statements you use to add the objects:

Object	Operation	SQL Statement
Catalog (all tables)	Add	CREATE CATALOG
Table	Add	CREATE TABLE
View	Add	CREATE VIEW
Index	Add	CREATE INDEX
Collation	Add	CREATE COLLATION
Partition	Add	ALTER TABLE PARTONLY MOVE ALTER INDEX PARTONLY MOVE
Column	Add	ALTER TABLE ADD COLUMN
Constraint	Add	CREATE CONSTRAINT
Comment	Add/Append	COMMENT

As with any change to the database, the first step is careful planning. Additional information on creating (adding) objects is located in Section 3, Understanding and Planning Database Tables, and Section 5, Creating a Database.

All changes to a database require the specified authority for protection of the database. The authority to add new SQL objects is given to anyone with authority to write to the catalog in which the object is registered. The authority to add to existing objects is controlled by the ownership and security of that object. For details on the authorization requirements, see Authorization Requirements for Database Operations on page 4-5.

Adding Catalogs

To add new catalogs to an existing data dictionary, use the CREATE CATALOG statement. Adding catalogs does not affect existing dictionary objects and items. The catalog format version of a new catalog is the same as the version of the SQL/MP software that creates the catalog, as explained in the *SQL/MP Version Management Guide*.

To create a new catalog and also secure the catalog in one statement, use the CREATE CATALOG statement with the SECURE option. For more information, see Altering Catalog Attributes on page 7-15. For general guidelines about the CREATE CATALOG statement, see Creating Catalogs on page 5-1.

To add a catalog:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Query the CATALOGS table of the system catalog to determine if the catalog name is available, or use the FILEINFO `catalog-name` TABLES command to verify that the catalog does not exist.

3. Enter the CREATE CATALOG statement, with or without the SECURE option.
4. Verify that the catalog's security is set to allow SQL objects to be added and accessed.

Adding Tables

Like adding new catalogs, adding new tables does not directly affect the existing dictionary objects, except for collations. Creating a table is the first step in defining SQL object dependencies, and no existing dependencies are affected. The new table can, however, be dependent on one or more collations.

You can add new tables to new catalogs or to existing catalogs. To add a table, use the CREATE TABLE statement.

If you are adding a new table to an existing application's database, you should ensure that the security and authority for the new table matches any existing security plan for allowing access.

If any column of the new table corresponds to an existing column in another table, you should define the new column with the same characteristics as the existing column.

A table added by a new version of SQL/MP software does not have the same version as the software unless the table uses one of the new features in that software version. For example, a table added by version 310 software is not a version 310 table unless it uses a feature not available in an older version of SQL/MP software. For more information, see the *SQL/MP Version Management Guide.*

To add a table, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Query the CATALOGS table of the system catalog or use the FILEINFO command to check that the target catalog already exists. The catalog must exist to create a table.
3. Query the TABLES table of the catalog to check that the table name is available.
4. Plan the column definitions, checking that the data type of any column that might be necessary for join or predicate search operations matches the joined column.
5. Make sure that collations exist before you refer to them in column definitions. If a collation to be used by a column does not exist, create the collation as explained under Creating Collations on page 5-55.
6. Enter the CREATE TABLE statement, or put the statement text into an EDIT file and enter an OBEY command to run the statement from the file.
7. Alter the security and ownership of the new table, if necessary.
8. For an audited table, make a TMF online dump. For a partitioned audited table, make an online dump of each partition.

For additional information and guidelines related to adding a table, see Creating Base Tables on page 5-10.

Adding Views

Adding views on existing tables does not affect existing database dependencies. To add a view, use the CREATE VIEW statement, following these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the names of any tables whose columns you want to include in the view. Shorthand views can also refer to other views.
3. Determine the column names of the view and the column names of the underlying table or tables of the view. To display the columns of the underlying table or tables, you can either use the INVOKE command or query the COLUMNS table of the catalog.
4. Enter the CREATE VIEW statement, or put the statement text into an EDIT file and enter an OBEY command to run the statement from the file.
5. Alter the security and ownership of the new view, if necessary.
6. For an audited view, make a new TMF online dump.

For additional information and examples of creating views, see Creating Views of Base Tables on page 5-38.

Adding Indexes

Use the CREATE INDEX statement to add an index to an existing table. The statement both creates and loads the new index.

Evaluating the Benefit of a New Index

Knowing when to add an index to improve performance requires a detailed understanding and analysis of your application. Following are the ways to collect data:

- Analyze the programs and ad hoc queries for the columns used in the DISTINCT, GROUP BY, ORDER BY, and WHERE clauses, and in join operations.
- Run the SQL compiler with the EXPLAIN option to obtain a report on the access paths the compiler chooses for the programs.
- Analyze Measure statistics on SQL statements.

If you need to determine whether an index can benefit performance, you could test the performance before implementing the index in the production system.

To test the effect of an index on performance, follow these steps:

1. Test a sample set of queries against the production tables by using the DISPLAY STATISTICS command to obtain the statistical information.
2. Duplicate the table or tables involved to a test location.
3. Create the new index.
4. Enter an UPDATE STATISTICS statement to update the statistical information stored in the catalog and get statistics on the new index. The CREATE INDEX statement does not automatically update statistical information.
5. SQL compile (recompile) any programs that use the table with the EXPLAIN option to determine whether the index is the chosen path.
6. Test the same queries against the tables by using DISPLAY STATISTICS to obtain the new statistical information.
7. Determine any improvement in performance.
8. If the query execution plans include using the new index and if you determine that the performance improvement is sufficiently advantageous over the increased system overhead of maintaining the index, add the index to the production database.
9. If you add the index, recompile programs that use the table.

Consider these guidelines when adding an index to an existing table:

- You should not create an index on a loaded table within a user-defined TMF transaction because the transaction could overflow the TMF audit trails and cause an error. The CREATE INDEX operation automatically initiates several TMF transactions, as necessary, but loads the index outside a TMF transaction. With this loading technique and the automatic transactions, the operation minimizes the TMF overhead of a potentially very long transaction and reduces output to the audit trails.
- Index creation can be a long operation, depending on the size of the table and the load on the system. Therefore, two locking strategies are available:
    - The default locking strategy acquires a shared table lock on the underlying table. The shared table lock ensures that no users can modify rows during the creation of the index. This lock can prohibit access to the table by other users that make write requests.
    - The WITH SHARED ACCESS option for the CREATE INDEX statement allows access to the table for DML operations during all but the short final stage of index creation. The option includes a reporting feature for monitoring index creation. In addition, you can request a time window or request explicit operator authorization for the final stage of index creation that requires table locking.

The WITH SHARED ACCESS option can be used in conjunction with the PARALLEL EXECUTION ON option if the initiating node, all nodes with base table partitions, and all nodes that will have index partitions are running version 315 or later of SQL/MP software.

For more information about the WITH SHARED ACCESS option and concurrent access to tables by multiple users, see Understanding the Implications of Concurrency on page 14-1 and the “WITH SHARED ACCESS” description in the *SQL/MP Reference Manual*.

- For an audited index, make a TMF online dump of the index immediately after creating it to prepare for possible file recovery, which might be faster than rebuilding the index.

Note. SQL tables and indexes with many partitions (typically around 400) might cause SQLCAT, SQLUTIL, or AUDSERV processes to incur file-system error 31 or 34 or cause the PARTNS catalog table and its associated index, IXPART01, to become full. For more information about this situation, see Creating Table Partitions on page 5-32.

For additional guidelines related to index creation, including performance-related considerations, see Creating Indexes on Base Tables on page 5-42.

Validation Considerations

Adding a new index invalidates the programs that depend on the underlying table unless you use one of these:

- The NO INVALIDATE option in the CREATE INDEX statement
- The CHECK INOPERABLE PLANS compiler option, described under Using Similarity Checks on page 10-15, with similarity checking enabled for the index

If you do not use one of the preceding options, you should include steps to explicitly SQL compile the dependent programs to avoid automatic recompilation and to return the application to a valid state.

The creation of the index does not automatically update the table’s statistics, which the SQL compiler uses to determine the best access path. You should always follow the creation of an index with the UPDATE STATISTICS statement to ensure that the table’s statistics are current. If the statistics are incorrect, the SQL compiler might not choose the most efficient access path.

Steps for Adding an Index

To add an index, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the table for which you want to add the index.
3. Optionally, determine which programs depend on this table by using the DISPLAY USE OF command. These programs will be invalidated unless you use one of the options discussed previously.
4. Optionally, prevent the use of the table for the duration of the CREATE INDEX operation to eliminate conflicts in access to the table; this operation requires exclusive use of the table during the final phase of the creation process.
5. For an index column, you can specify a different collation than the collation used by the corresponding base table column, provided the shifting rules of both collations are the same. Make sure that collations exist before you refer to them in column definitions. If a collation to be used by a column does not exist, create the collation as explained under Adding Collations on page 7-13.
6. Enter the CREATE INDEX statement. To allow an online dump during index creation, use the WITH SHARED ACCESS option.
7. Enter the UPDATE STATISTICS statement for the underlying table.
8. SQL compile the invalidated programs to enable the compiler to determine the best access strategy.
9. For audited indexes, make a new TMF online dump.
10. Restart use of the table if you stopped its use.

To maximize concurrent access during the index creation operation, use the WITH SHARED ACCESS option. For more information, see Understanding the Implications of Concurrency on page 14-1.

Adding Partitions to Tables and Indexes

To add a new partition to a key-sequenced table, use the ALTER TABLE statement with the PARTONLY MOVE specification. To add a new partition to an index, use the ALTER INDEX statement. Alternatively, you can split partitions, merge partitions, or move row boundaries within existing partitions.

You can also use the ADD PARTITION option with the ALTER TABLE or ALTER INDEX statement to add a new partition to a table or index. The ADD PARTITION option is equivalent to the one-way split form of the PARTONLY MOVE option. However, to use enhanced features such as the WITH SHARED ACCESS option, you must use the PARTONLY MOVE option.

For a relative or entry-sequenced table, the only way to add a new partition is to add an empty partition to the end of a table with the ADD PARTITION option. You cannot use the PARTONLY MOVE option with a relative or entry-sequenced table.

Evaluating the Benefit of a New Partition

Partitioning might increase performance in these cases:

- If disk accesses are queued in the disk process, partitioning the table across multiple volumes might increase performance.
- If a partition of a table or index is full, the partition can be split into two partitions.
- If a certain subset of a remote table’s data is accessed more frequently at the local node than from a remote node, partitioning the table so that the frequently accessed portion of the data resides on the local node can increase local performance.
- If a distributed table residing on a remote node might be frequently unavailable, partitioning the table so that the local partition can be accessed regardless of the remote table’s availability can increase local performance.
- If an application program that disables row lock escalation to table locks receives an error from the disk process because the disk process has used up the control block space for locks, the table can be partitioned to allow more locks. Partitioning allows more locks to be placed on a table because the disk process lock limit functions on a partition basis.
- If queries are processed in parallel, partitioning a table or index is often required. Partitioning is necessary, for example, for parallel execution of a SELECT statement on a single table. Also for join queries, which do not require partitioning of the objects involved, parallel processing operates best when a table or index is partitioned.

You can use the Measure product to obtain statistics concerning disk message levels, queuing, and other measurements on various volumes or file partitions to identify the levels of use.

Steps for Adding a Partition

You can add partitions to tables and indexes within the guidelines listed in the *SQL/MP Reference Manual* in the descriptions of the ALTER TABLE and ALTER INDEX statements.

To add a partition, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the table or index to which you want to add the partition.
3. For a key-sequenced table or index, determine the starting key of the new partition.

4. If you are splitting a partition, check that ample disk space is available for the new partition. For information about space requirements, see the ALTER TABLE or ALTER INDEX statement in the *SQL/MP Reference Manual*.

5. Use the DISPLAY USE OF command to determine which programs depend on the table. These programs will be invalidated unless the programs are compiled with the CHECK INOPERABLE PLANS option and the table has similarity checking enabled, as described under Using Similarity Checks on page 10-15.

6. If you wish to add Format 2 partitions to a Format 2-enabled table, enter the ALTER TABLE or ALTER INDEX statement with the FORMAT clause to specify the format of the new partition. For more information about Format 2 partitions, see Appendix C, Format 2 Partitions.

7. Enter the ALTER TABLE or the ALTER INDEX statement with the PARTONLY MOVE clause to add the partition.

8. SQL compile the invalidated programs identified by the DISPLAY USE OF command in Step 5.

9. For an audited table or index, make new TMF online dumps of all affected partitions.

   Note that the new partition inherits the partition array value associated with the base table. If PARTITION ARRAY is EXTENDED, the partition can make use of the larger number of partitions available for versions 320 and later of SQL/MP software. If PARTITION ARRAY is FORMAT2ENABLED, the partition can make use of the larger size of partitions available for versions 350 and later of SQL/MP software. In Step 6, use the FORMAT clause to specify whether the added partition should be Format 1 or Format 2. Partitions added to Format 2-enabled tables and indexes will be Format 2 partitions by default.

Note. SQL tables and indexes with many partitions (typically around 400) might cause SQLCAT, SQLUTIL, or AUDSERV processes to incur file-system error 31 or 34 or cause the PARTNS catalog table and its associated index, IXPART01, to become full. For more information about this situation, and for general information about adding a partition, see Creating Table Partitions on page 5-32.

For information about redistributing rows across partitions, see Splitting, Moving, and Merging Partitions on page 7-20.

Example

This example adds an empty partition to a key-sequenced table, leaving existing data in the existing partition (assuming there are no key values past 4999). In this one-way split operation, the starting key value for the new partition is 5000.

```
>>  ALTER TABLE $VOL1.SALES.CUSTOMER
+>     PARTONLY MOVE FROM KEY 5000 TO $VOL3.SALES.CUSTOMER
+>     CATALOG $VOL1.SALES
+>     EXTENT (1000,200);
--- SQL operation complete.
```

Adding Columns

You can add a column to any key-sequenced table. In addition, you can add a column to any relative table already defined with enough extra bytes in the RECLENGTH value to accommodate the new column. You cannot, however, add columns to entry-sequenced tables or to views or indexes. To add a column, use the ALTER TABLE statement with the ADD COLUMN clause. Each ALTER TABLE statement adds only one column. To add several columns, use the statement once for each column.

Each new column is added as the last column in the table. If you want to add a column to a table so that it does not appear as the last column, follow the steps for altering columns under Altering Database Objects on page 7-13.

Existing programs that depend on the table are not affected by the addition of a new column unless a program needs to use the new column or includes an INSERT * or a SELECT * statement that refers to the new column's table. Adding a column, however, invalidates programs that depend on the table unless the program was compiled with the CHECK INOPERABLE PLANS option and the similarity check is enabled for the table. For more information, see Using Similarity Checks on page 10-15.

You should include steps to explicitly SQL compile invalid programs to avoid automatic recompilation and to return the application to a valid state.

Adding a column does not cause any existing data to be rewritten. For existing rows, the new column takes on the system default value unless you specify a default value. Adding a column to a table does not affect existing dependent views or indexes.

Note. Views previously defined as "SELECT * FROM..." will not select the new column.

The ALTER TABLE statement with the ADD COLUMN clause requires an exclusive table lock to ensure that no rows are inserted during creation of the column. All partitions and protection views must also be accessible.

After you add a column, you should consider whether the new column must be integrated into existing or new views and indexes. Application programmers can write new programs to use the new column or alter existing programs to use the new column.

This example adds a column to the CUSTOMER table:

```
>>  LOG $VOL.DBCHANGE.CNGLOG;
>>  ALTER TABLE $VOL1.SALES.CUSTOMER
+>     ADD COLUMN PRIOR_YEARS_SALES
+>             PIC S9(9)V99   COMP   DEFAULT SYSTEM NOT NULL;
--- SQL operation complete.
```

To add a column, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the table to which you want to add the column. You can add columns to all key-sequenced tables or to relative tables with sufficient RECLENGTH length.
3. Match the data type for the column with any corresponding column’s data type so that users can perform joins or predicate searches.
4. Determine which programs depend on the table by using the DISPLAY USE OF command. These programs will be invalidated unless a program was compiled with the CHECK INOPERABLE PLANS option and the similarity check is enabled for the table. For more information, see Using Similarity Checks on page 10-15.
5. Optionally, prevent the use of the table for the duration of the ALTER TABLE operation to eliminate conflicts in access to the table; this operation requires exclusive use of the table.
6. Enter the ALTER TABLE statement with the ADD COLUMN clause.
7. Determine if the new column will also be added to any existing index or view or if program changes are required. After adding the new column, follow the steps for integrating the new column into an existing application, described in the following text.
8. SQL compile the invalidated programs.
9. For an audited table, make a new TMF online dump.
10. Restart use of the table if you stopped its use.

To integrate the new column into the existing database or application programs, do these:

- If you want to create a new index using the new column, follow the steps for adding indexes.
- If you want to add the new column to an existing index, first follow the steps for dropping indexes, then follow the steps for adding indexes to add the new column definition. You cannot alter an index or view to add a column.
- If you want to create a new view using the new column, follow the steps for adding views.
- If you want to add the new column to an existing view, first follow the steps for dropping views, then follow the steps for adding views. You cannot alter a view to add a column.
- If you want to use the new column in programs, you must change existing programs to refer to the new column. You might need to change screen sections to display the column on the screen, and you might need to change code sections to

retrieve or update the column. After you SQL compile a program changed to use the new table definition, the program can use the column.

- If you want to add constraints that control values in the new column or if you want to add comments on the new column, you can run the CREATE CONSTRAINT or COMMENT statement at any time after the column is added.

For additional information about column positioning and performance-related aspects of columns, see Defining Columns on page 5-19.

Adding Constraints

Adding a constraint to the database is similar to making a program change. Any future data insertions or updates must satisfy the new rule imposed by the constraint. In addition, all existing rows must satisfy the rule before a constraint can be added. To add a constraint, use the CREATE CONSTRAINT statement.

Adding a constraint to a table invalidates any programs that depend on the table. You should include steps to explicitly SQL compile the dependent programs to avoid automatic recompilation and to return the application to a valid state.

To add a constraint, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the table for which you want to add the constraint.
3. Check any existing constraints by querying the CONSTRNT catalog table and determine whether the new constraint would supersede or conflict with any constraint already defined for the same table.
4. Enter a query on the table by making the predicate of the query the negation of the constraint. This query would identify rows that do not satisfy the constraint. If the query identifies any rows, change or delete the rows before creating the constraint. If the query refers to a collation, verify that the collation exists and is secured to allow you write access before creating the constraint.
5. Determine which programs depend on the table by using the DISPLAY USE OF command. These programs will be invalidated.
6. Optionally, prevent the use of the table for the duration of the CREATE CONSTRAINT operation to eliminate conflicts in table access; this operation requires exclusive use of the table.
7. Enter the CREATE CONSTRAINT statement.
8. SQL compile the invalidated programs.
9. Restart use of the table if you stopped its use.

For additional information on constraints and related performance issues, see Creating Constraints on Data on page 5-51 and Checking Data Integrity on page 14-23.

Adding Collations

Adding collations to a database does not affect existing database dependencies. To add a collation, use the CREATE COLLATION statement, following these steps:

1. Create the collation source file, as explained under Creating Collations on page 5-55.
2. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
3. Enter the CREATE COLLATION statement, or put the statement text into an EDIT file and enter an OBEY command to run the statement from the file.
4. Alter the security and ownership of the new collation if necessary.
5. For more information about collations, see Creating Collations on page 5-55.

Adding Comments

Comments are allowed for these objects: a column, table, view, constraint, or index, table or view column, or collation. To add comments, use the COMMENT statement.

You can add comments to clarify how the object is used. Having comments in the active data dictionary can help both database administrators and programmers in understanding the database structure. Application users would not normally use the dictionary comments.

You can add or append comments to any existing text. If you use the CLEAR clause, the new comment overwrites any existing comments on the specified object.

To add a comment, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the object for which you want to add a comment.
3. Determine if any comments exist and whether you want the new comment to be appended or to overwrite existing comments.
4. Enter the COMMENT statement.

Altering Database Objects

The active data dictionary enables you to make certain changes to the database online. As with any change to the database, careful planning should be the first step.

To alter the security and physical file attributes of SQL objects, use the ALTER statement. To alter the security attributes of tables, views, collations, and SQL programs stored in Guardian files, use the SECURE utility. To alter the security attributes of SQL programs stored in OSS files, use the appropriate OSS utility. For more information, see the *Open System Services Shell and Utilities Reference Manual*.

All changes to a database require the specified authority for protection of the database. The authority to alter existing objects is controlled by the ownership and security of that object. For details on authorization requirements, see Authorization Requirements for Database Operations on page 4-5.

This table summarizes the database objects that can be altered and the statements, commands, and options you use for the operations:

Object	**Operation**	**Statement or Command** (page 1 of 2)
Catalog (all tables)	Security	ALTER CATALOG
Catalog tables PROGRAMS USAGES TRANSIDS CATALOGS	Security	ALTER TABLE
Table	Security/attributes	ALTER TABLE SECURE
View	Security	ALTER VIEW SECURE
	Column attributes	ALTER VIEW COLUMN
	Heading text	ALTER VIEW COLUMN
	Help text	HELP TEXT
Index	File attributes	ALTER INDEX
Partition	File attributes	ALTER TABLE PARTONLY ALTER INDEX PARTONLY
	Add/split/move	ALTER TABLE PARTONLY MOVE ALTER INDEX PARTONLY MOVE
	Drop	ALTER TABLE DROP PARTITION ALTER INDEX DROP PARTITION
Column	New definition	CREATE TABLE LOAD DROP TABLE
	Column attributes	ALTER TABLE ADD COLUMN
	Heading text	ALTER TABLE COLUMN
	Help text	ALTER TABLE COLUMN HELP TEXT

Object	**Operation**	**Statement or Command** (page 2 of 2)
Constraint	New definition	DROP CONSTRAINT CREATE CONSTRAINT
Collation	Security	ALTER COLLATION SECURE
	Rename	ALTER COLLATION RENAME
Comment	Add/append/drop	COMMENT

Altering Catalog Attributes

Use the ALTER CATALOG statement to change the security of an entire set of catalog tables. You can alter only the security specifications of a catalog. This statement does not affect the system catalog CATALOGS table. You must alter that table with the ALTER TABLE statement.

You cannot specify a security string for a catalog that does not include read access by the owner of the catalog. Requiring that the catalog has read access by the owner ensures that the owner of the catalog can subsequently read it.

The PROGRAMS, USAGES, and TRANSIDS tables of an SQL catalog and the CATALOGS table of the system catalog can be secured separately from the remainder of the catalog tables. To resecure these tables, you can use the ALTER TABLE statement as described next under Altering Table Attributes on page 7-15.

This example alters the security of all the catalog tables of the catalog PERSNL:

```
>>  ALTER CATALOG PERSNL SECURE "NNNO";
--- SQL operation complete.
```

To alter the security of a catalog, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the catalog name and existing security.
3. Enter the ALTER CATALOG statement.

For more information about defining catalog tables, see Creating Catalogs on page 5-1.

Altering Table Attributes

To alter the security and physical file attributes of SQL tables, use the ALTER statement. To alter the security attributes of tables, use the SECURE utility.

Altering a table's attributes or security specification neither invalidates any programs nor affects dependencies of the table. If you alter the security of the table, however, you might damage the security scheme of the dependent views and the access strategy. If you alter the audit flag, you can invalidate the most recent TMF online dump, and programs expecting an audited table will receive a TMF run-time error.

You can alter a single partition of a partitioned table by specifying the PARTONLY clause in the ALTER statement. For a partitioned table, if you omit PARTONLY, the statement operates on all partitions, and all partitions must be accessible.

These examples demonstrate altering the security and file attributes of a table:

```
>>  ALTER TABLE $VOL1.PERSNL.EMPLOYEE OWNER 100,001
+>   SECURE "NGOO";
--- SQL operation complete.
>>  ALTER TABLE $VOL1.SALES.ORDERS MAXEXTENTS 300;
--- SQL operation complete.
>>  ALTER TABLE $VOL1.SALES.CUSTOMER NOPURGEUNTIL DEC 31 1990;
--- SQL operation complete.
```

Use the ALTER TABLE statement with the COLUMN specification to add or change heading text for an existing column of a table. Use the HELP TEXT statement to add or change help text for a column. It is not possible to alter the other attributes or the data type of a column.

Use the ALTER TABLE statement with the SIMILARITY CHECK ENABLE clause to enable similarity checking for a table.

This example shows how to alter the heading text for the column EMPNUM of the EMPLOYEE table. If the column did not previously have a heading, the new heading is added. If the column previously had a heading, the old heading is replaced by the new one.

```
>>  ALTER TABLE $VOL1.PERSNL.EMPLOYEE COLUMN EMPNUM
+>     HEADING "Employee ID Number";
--- SQL operation complete.
```

Specifying NO HEADING in the ALTER TABLE COLUMN statement deletes any existing heading text from the column.

Altering the heading text for a table column does not update any dependent views created with the headings of the underlying table columns. To keep the two objects synchronized, you must update the view’s column headings independently.

You can also alter the partition array type associated with a table. If you change the array type, all programs that refer to the table are invalidated. In addition, if you modify the type from EXTENDED to STANDARD, the data structures might not fit within the STANDARD format. When this situation occurs, SQL returns an error.

Steps for Altering Table Attributes

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the table you want to alter.
3. If you are altering security or the audit attribute, determine whether the change will affect current users or programs.
4. Enter the ALTER TABLE statement.

For more information about table columns and attributes, see Creating Base Tables on page 5-10.

Altering Security

To make the USAGES, TRANSIDS, and PROGRAMS tables accessible for SQL compilations of programs, you might need to change the security of each table in an ALTER TABLE statement. During explicit SQL compilation, any dependencies a program has on tables or views described in a catalog are recorded in the catalog's USAGES table. To insert the dependency record into the USAGES table, the catalog manager must start a TMF transaction that is registered in the TRANSIDS table. Write access to the PROGRAMS table is required so that the SQL compiler can register programs in the table.

You can change the catalog security at creation time by specifying the SECURE attribute in the CREATE CATALOG statement. You can also change the security of these individual tables at any later time by using the ALTER CATALOG statement:

- CATALOGS (system catalog only)
- USAGES
- TRANSIDS
- PROGRAMS

If you use the SECURE attribute, you must specify a security string that gives the owner of the catalog tables read access.

For a user to compile a program, the user needs read and write access to the USAGES and TRANSIDS tables in a catalog containing descriptions of tables, views, collations, partitions, and indexes that the program uses. Additionally required is write access to the PROGRAMS table of the catalog in which the program is registered.

The catalog tables compose the data dictionary, a vital part of an application's integrity. The security of a catalog should protect the data dictionary information from unauthorized removal or alteration.

Altering View Attributes

You can alter the security string specification of a view but not the attributes. You can alter the owner ID for a shorthand view but not for a protection view. To alter the security or owner ID, use the ALTER VIEW statement.

Altering a view's security neither invalidates any programs nor affects the dependent views. If you alter the security of the view, however, you might damage the security scheme of the dependent views and the access strategy. You can alter the security of a view by using the ALTER VIEW statement or the SECURE command. For a detailed description of view security dependencies, see the description of the ALTER VIEW statement in the *SQL/MP Reference Manual*.

This example alters the security attributes of a shorthand view:

```
>>  ALTER VIEW $VOL1.PERSNL.NAMELIST OWNER 100,001
+>      SECURE "NNNO";
--- SQL operation complete.
```

You can create a view that inherits the heading text or help text from the underlying table. Alternatively, you can create new headings and help text for the columns of the view. After the view is created, you can also alter the heading text or help text as an independent operation.

To add or alter the heading text for a column, use the ALTER VIEW statement. To add or alter help text for a column, use the HELP TEXT statement. Altering the heading text or help text of the columns in the underlying table does not alter the heading text or help text inherited by the view.

Use the ALTER VIEW statement with the SIMILARITY CHECK ENABLE clause to enable similarity checking for a protection view.

This example demonstrates altering the heading text for the column EMPNUM of the EMPLIST view. If the column did not previously have a heading, the new heading is added. If the column already had a heading, the old heading is replaced by the new one.

```
>>  ALTER VIEW $VOL1.PERSNL.EMPLIST COLUMN EMPNUM
+>      HEADING "Employee ID Number";
--- SQL operation complete.
```

To alter a view, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the view you want to alter and the name of the table that underlies the view.
3. Determine whether the change meets the security dependencies of dependent views.
4. Enter the ALTER VIEW statement.

For more information about view attributes, see Creating Views of Base Tables on page 5-38.

Altering Index Attributes

The ALTER INDEX statement can alter several file and security attributes of an index. For security attributes, you can alter only CLEARONPURGE, NOPURGEUNTIL, or SECURE. The index owner and security are set and altered by those attributes of the underlying table. You can independently alter all file attributes of an index except the AUDIT attribute.

You can alter a single partition of a partitioned index by specifying the PARTONLY clause in the ALTER INDEX statement.

For a partitioned index, if you omit PARTONLY, the statement operates on all partitions, and all partitions must be accessible.

For a detailed description of index security dependencies, see the description of the ALTER INDEX statement in the *SQL/MP Reference Manual*.

These examples alter attributes of an index:

```
>>  ALTER INDEX $VOL1.PERSNL.XEMPL NO CLEARONPURGE;
--- SQL operation complete.
>>  ALTER INDEX $VOL1.SALES.XORDCUS
+>      PARTONLY MAXEXTENTS 300;
--- SQL operation complete.
```

To alter an index, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the index you want to alter.
3. Enter the ALTER INDEX statement.

For more information about index attributes, see Creating Indexes on Base Tables on page 5-42.

Altering Partition Attributes

You can alter the attributes of a single partition of a partitioned table or index by specifying the PARTONLY clause in the ALTER TABLE or ALTER INDEX statement.

You can alter the partition attributes MAXEXTENTS, ALLOCATE, and DEALLOCATE. You cannot alter the security string for a partition.

This example deallocates unused extents of a table partition located on $VOL1.SALES:

```
>>  ALTER TABLE $VOL1.SALES.ORDERS
+>      PARTONLY DEALLOCATE;
--- SQL operation complete.
```

This example sets the maximum number of extents for a partition of an index located on $VOL1.SALES:

```
>>  ALTER INDEX $VOL1.SALES.XORDCUS
+>      PARTONLY MAXEXTENTS 300;
--- SQL operation complete.
```

Splitting, Moving, and Merging Partitions

You can move rows within partitions of a base table or index. To do this, use the PARTONLY MOVE option with the ALTER TABLE or ALTER INDEX statement. You can perform these operations that split, move, or merge partitions:

- Move a partition from one volume to another
- Perform a one-way partition split
- Perform a two-way partition split
- Move partition boundaries—move rows from one partition into another existing partition
- Merge a partition into another existing partition

For many of these operations, you can specify the WITH SHARED ACCESS option to retain full read and write access to data throughout most of the operation. (Some operations require the WITH SHARED ACCESS option.)

For a description of the steps you should follow when performing these operations, see Steps for Adding a Partition on page 7-8.

Moving a Partition to Another Volume

You can move a partition to another volume with or without using the WITH SHARED ACCESS option. If a table or index is not partitioned, you can use the MOVE option without the PARTONLY clause to move the entire table or index to the new volume. You can perform this move to change the format of a partition.

After the partition is moved to the target volume, SQL automatically drops the original partition (from the source volume).

This example moves a partition of the EMPLOYEE table to another volume while keeping the partition available for updates and reads during most of the operation:

```
>>  ALTER TABLE $VOL5.PERSNL.EMPLOYEE
+>     PARTONLY MOVE TO $VOL10.PERSNL.EMPLOYEE
+>     CATALOG $VOL1.PERSNL
+>     EXTENT (1000,200)
+>     WITH SHARED ACCESS;
--- SQL operation complete.
```

Performing a One-Way Partition Split

A one-way split moves data in the first or last part of a partition into a new partition. The remaining part of the data stays in the original partition. You can perform a one-way split with or without using the WITH SHARED ACCESS option.

You can perform this move to change the format of a partition.

If you do not use the WITH SHARED ACCESS option, you can only move data in the last part of the partition to a new partition; you cannot move data in the first part of the partition. That is, you can only use the FROM KEY *value* clause with the PARTONLY MOVE option; you cannot use the UP TO KEY *value* clause. (You can optionally use the UP TO LAST KEY clause, which indicates that SQL should move the data from the FROM KEY *value* up to the last key value in the partition.)

If you do use the WITH SHARED ACCESS option, you can specify either the first or last part of the data during the operation. That is, you can use either the FROM KEY *value* clause or the UP TO KEY *value* clause with the PARTONLY MOVE option.

This example splits an existing partition of an index. Before the split operation, the index has three partitions with these starting key values:

Partition Location	Starting Key Value
$VOL1	0
$VOL2	10000
$VOL3	20000

In this one-way split operation, the starting key value for the new index partition is 5000. The new partition resides on $VOL4. Rows with index key values that equal or exceed 5000, but that are less than the starting key value assigned to the next numeric partition (that is, rows with key values from 5000 to 9999) are relocated to the new partition:

```
>>  ALTER INDEX $VOL1.SALES.CUSTNAME
+>     PARTONLY MOVE FROM KEY 5000 TO $VOL4.SALES.CUSTNAME
+>     CATALOG $VOL1.SALES
+>     EXTENT (1000,200);
--- SQL operation complete.
```

After the split operation, the index has four partitions with these starting key values:

Partition Location	Starting Key Value
$VOL1	0
$VOL4	5000
$VOL2	10000
$VOL3	20000

This example shows how to move the latter portion of one partition into a new partition (a one-way split), using a define for the index name:

```
>> ALTER INDEX =XPART_LOC
+>   PARTONLY MOVE
+>     FROM KEY “H00” UP TO LAST KEY TO =XPART_EUROPE
+>     CATALOG =INVENT_EUROPE
+>   EXTENT (8,8) SLACK 20;
--- SQL operation complete.
```

The preceding example uses the FROM KEY *value* clause together with the UP TO LAST KEY clause to specify the last part of the partition. (The LAST KEY refers to the last key value in the partition, not the entire file.) The UP TO LAST KEY clause is optional; if you omitted it, the preceding example would move the same rows (from H00 to the last key in the partition) into the new partition.

Performing a Two-Way Partition Split

A two-way split moves all the data from an existing partition into two new partitions. After a two-way split, SQL automatically drops the original partition. The data is now divided between the two new, different partitions. You cannot use the WITH SHARED ACCESS option with a two-way split operation.

You can perform this move to change the format of a partition.

This example shows how to split an existing partition into two new partitions and register the new partitions in other catalogs (a two-way split):

```
>>ALTER TABLE $DISK1.SALES.ORDERS
+>   PARTONLY MOVE
+>     ( FROM FIRST KEY UP TO KEY 50 TO $DISK2 CATALOG =CAT2,
+>       FROM KEY 50 UP TO LAST KEY TO $DISK3 CATALOG =CAT3 );
--- SQL operation complete.
```

The preceding example specifies the two destination volumes with the volume names only ($DISK2 and $DISK3). You do not have to specify the subvolume and file names because these names must be the same for each partition in the table or index. (If you omit the node name, SQL defaults to the local node.)

Moving Partition Boundaries

You can move the boundaries between two existing partitions—that is, move rows from one partition to another—by using the PARTONLY MOVE option. A move partition boundary operation moves data in the first or last part of a partition into the logically adjacent partition. You must use the WITH SHARED ACCESS option to perform this operation.

During this operation, SQL moves the specified rows and automatically adjusts the key ranges of the two affected partitions. The move partition boundary operation is similar to a one-way split, but it moves data into an existing partition instead of a new partition.

Suppose, for example, that a partition residing on $DISK2 contains rows in the key range 3000 through 5999, and a partition residing on $DISK3 contains the key range 6000 through 8999. Suppose further that users have inserted many more rows into lower key range than the higher one, and that you want to adjust the partition boundaries accordingly.

This example moves rows from the partition residing on $DISK2 into the adjacent partition residing on $DISK3. SQL moves the rows starting with the value 5000 up to the last row in the partition.

```
>>ALTER TABLE $DISK2.SALES.ORDERS
+>   PARTONLY MOVE
+>       FROM KEY 5000 UP TO LAST KEY TO $DISK3
+>       CATALOG $DISK1.SALES
+>       WITH SHARED ACCESS;
--- SQL operation complete.
```

The preceding example specifies the table by naming the exact partition from which rows will be moved (ALTER TABLE $DISK2.SALES.ORDERS). In fact, you can specify any partition to identify the table; SQL will move rows from the correct partition based on the FROM KEY *value* clause you specify.

Merging Partitions

A merge operation moves all the data in a partition into an existing, logically adjacent partition. Merging partitions is a form of moving partition boundaries, except that the merge operation moves all rows from the original partition into the destination partition. After a merge operation, SQL automatically drops the original partition.

A merge operation is also similar to a move operation, except that it moves the data into an existing partition instead of a new partition. You must use the WITH SHARED ACCESS option with a merge operation.

This example moves all rows from the partition of the CUSTOMER table residing on $VOL10 into the existing partition residing on $VOL11. The two partitions are logically adjacent:

```
>>ALTER TABLE $VOL10.SALES.CUSTOMER
+>   PARTONLY MOVE TO $VOL11
+>       CATALOG $VOL1.SALES
+>       WITH SHARED ACCESS;
--- SQL operation complete.
```

To perform a merge operation, you must specify the exact partition being merged in the ALTER TABLE statement. The merge operation does not require you to specify a key range because, like a move operation, it moves all the rows in the partition. In the preceding example, suppose that he FIRST KEY value of the partition residing on $VOL10 is 5000, and the FIRST KEY value of the partition on $VOL11 is 10000. After the merge operation, the FIRST KEY of the partition residing on $VOL11 is 5000, and the partition on $VOL10 no longer exists.

UNRECLAIMED FREESPACE (F) and INCOMPLETE SQLDDL OPERATION (D) Flags

A split partition, merge partition, or move partition boundary operation can cause these flags to be set:

- UNRECLAIMED FREESPACE (F) indicates that an SQL object contains unusable space. This flag is set as follows:
  - For the source partition after a move boundary or one-way split operation using the WITH SHARED ACCESS option completes successfully. Move boundary operations perform a special partial PURGEDATA request to eliminate the data from the source partition copied to the target partition. The UNRECLAIMED FREESPACE flag indicates that space must be deallocated in the partition. Applications can continue to read and update the table or index, even though the UNRECLAIMED FREESPACE flag is set, but subsequent ALTER operations and BACKUP, DUP, or move operations might fail.

    To reclaim the space, issue a FUP RELOAD operation for the source partition. After the UNRECLAIMED FREESPACE flag is reset, you can stop the FUP RELOAD operation if desired and resume normal activity on the table or index. (The FUP RELOAD operation reclaims free space first. If you do not stop the FUP RELOAD operation, it continues with a file reorganization step.) For more information about stopping a FUP RELOAD operation, see Suspending a Reorganization Operation on page 8-4.
  - For the target partition, if a move boundary request fails after data is loaded into the target partition. In this situation, there is free space in the target partition.

    To reclaim the space, issue a FUP RELOAD operation for the target partition before rerunning the ALTER operation.
- INCOMPLETE SQLDDL OPERATION (D) indicates one of these two situations:
  - A move partition boundary or merge partition operation using the WITH SHARED ACCESS option is in progress. The INCOMPLETE SQLDDL OPERATION flag is set for the target partition during the operation and is reset when the operation completes successfully.
  - If a merge partition or move partition boundary operation does not complete successfully because of a processor failure or another reason, the INCOMPLETE SQLDDL OPERATION flag remains set. In this instance, the object might contain invalid data and be in a corrupted state.

    To recover from this situation, issue an ALTER statement with the RECOVER INCOMPLETE SQLDDL OPERATION option. After recovering the table or index, check the UNRECLAIMED FREESPACE flag to see if a FUP RELOAD operation is needed. Avoid using the object while the INCOMPLETE SQLDDL OPERATION flag is set; the data cannot be assumed to be accurate.

Operational Considerations Related to F and D Flags

If an SQL object has the UNRECLAIMED FREESPACE or INCOMPLETE SQLDDL OPERATION flag set, reset the flag as described previously before backing up, moving, or duplicating the object. Otherwise, these situations occur:

- If you attempt to use SQLCI DUP on an object that has either flag set, SQL returns an error.
- If you attempt to use BACKUP on an object with the UNRECLAIMED FREESPACE flag set, SQL returns a warning and backs up the object.

  A subsequent RESTORE of the object generates a warning, and SQL restores the object without the UNRECLAIMED FREESPACE flag set.

△ **Caution.** Do not run FUP RELOAD to recover the free space in the object; you might cause a processor to fail.

  Use the FILCHECK utility to determine if unreclaimed free space exists in the object. If the object has unreclaimed free space, use one of these actions to correct the problem:

  - Create a new SQL object similar to the one with unreclaimed free space and use the SQL COPY or LOAD command to load data from the defective object.
  - Use the SQLCI ALTER TABLE <name> PARTONLY MOVE statement to create a new partition for the data.

- If you attempt to use BACKUP for the primary partition of an object or for a secondary partition (using the PARTONLY attribute), and the INCOMPLETE SQLDDL OPERATION flag is set for the object, one of this occurs:

  - If the IGNORE option is not specified, SQL returns an error and does not back up the object.
  - If the IGNORE option is specified, SQL returns a warning and backs up the object.

  A subsequent RESTORE of the object operates as follows:

  - If the IGNORE option is not specified, SQL returns an error and does not restore the object.
  - If the IGNORE option is specified, SQL returns a warning and restores the object. The newly-restored object does not have the INCOMPLETE SQL DDL OPERATION flag set. At this point, there is no way to determine whether this flag had been set.

  Note. In the current version of software, if you attempt to use BACKUP for an entire partitioned table or index and a secondary partition has the INCOMPLETE SQLDDL OPERATION flag set, SQL backs up (and restores, if requested) the entire table or index regardless of the flag setting.

If there is a concern that a restored table might have had the INCOMPLETE SQLDDL OPERATION flag set, use the SQLCI ALTER TABLE <name> PARTONLY RECOVER INCOMPLETE SQLDDL OPERATION command for the table. This step will not harm the table, even if it did not have the flag set previously.

- If you attempt to use SQL DUP from a node running version 315 or earlier, SQL does not recognize the flag and proceeds with the operation. If the UNRECLAIMED FREESPACE flag is set, a FUP RELOAD of the object from a node running a SQL/MP version earlier than 315 might corrupt the object or cause a processor halt. If the INCOMPLETE SQLDDL OPERATION flag is set, the target object might have extraneous data that will be visible to accessing applications. Do not run UPDATE STATISTICS when one of these flags is set. The results might be incorrect.

For more information about partitions, see Creating Table Partitions on page 5-32 and Creating Index Partitions on page 5-48.

Altering Columns

You are not allowed to alter column definitions or sizes. To achieve an alteration of an existing column, you must first create a new table with the column sizes and data type definitions you want and then load the new table from the old table.

You cannot alter a view or index definition to add or delete columns. You can accomplish these operations by dropping the old object and adding a new object to comply with the new structure.

Also, you cannot change a collation associated with a column. You can, however, add an index with a different collation for the column, provided both collations have the same shifting rules.

Example 7-1 on page 7-27 shows the operations required to effectively alter a column. The column CUST_PO is increased from 35 to 45 bytes. The example includes the original table definition ($VOL1.SALES.ORDERS), the CREATE TABLE statement for the new table, and the LOAD command. The LOAD command must include the MOVEBYNAME option to load the new table correctly.

Example 7-1. Altering a Column

```
* Record Definition for $VOL1.SALES.ORDERS
 01 ORDERS.
    02 ORDERNUM                      PIC 9(6).
    02 ORDER_DATE                    PIC S9(6) COMP.
    02 DELIV_DATE                    PIC S9(6) COMP.
    02 SALESREP                      PIC 9(4).
    02 CUST_PO                       PIC X(35).
    02 CUSTNUM                       PIC 9(4).

>> LOG $VOL1.DBCHANGE.CNGLOG;
>> CREATE TABLE $VOL1.SALES.ZZORDERS
+>    (ORDERNUM             PIC 9(6)  NO DEFAULT NOT NULL,
+>     ORDER_DATE           PIC S9(6) COMP DEFAULT SYSTEM NOT
NULL,
+>     DELIV_DATE           PIC S9(6) COMP DEFAULT SYSTEM NOT
NULL,
+>     SALESREP             PIC 9(4)  DEFAULT SYSTEM,
+>     CUST_PO              PIC X(45) DEFAULT SYSTEM NOT NULL,
+>     CUSTNUM              PIC 9(4) DEFAULT SYSTEM NOT NULL,
+>    PRIMARY KEY ORDERNUM)
+>    EXTENT (1000,100)
+>    CATALOG $VOL1.SALES;
--- SQL operation complete.
>> ALTER TABLE $VOL1.SALES.ZZORDERS NO AUDIT;
--- SQL operation complete.
>> LOAD $VOL1.SALES.ORDERS, $VOL1.SALES.ZZORDERS,
+>    MOVEBYNAME;
--- SQL operation complete.
>> ALTER TABLE $VOL1.SALES.ZZORDERS AUDIT;
--- SQL operation complete.
```

For more information about column attributes, see Defining Columns on page 5-19.

Altering Constraints

You cannot alter constraints, but you can change them by dropping an existing constraint or adding a new constraint to the table. Constraints reside in definition only; therefore, they have no physical or security attributes to alter.

Altering Collation Attributes

You can alter the collation's owner or security string by using the ALTER COLLATION statement. To alter a collation, you must have authority to read and write to the collation and the catalog in which the collation is registered.

△ **Caution.** Altering the security of a collation might restrict access to objects and programs that use the collation.

If you alter the security of a collation, be careful not to restrict access for dependent objects and programs. Altering collation security also alters the security of all objects and programs that use the collation.

This example changes the security of a collation to allow all network users access to the collation:

```
>>  ALTER COLLATION $VOL1.SALES.SPANISH
+>      SECURE “NU-U”;
--- SQL operation complete.
```

For more information about collation attributes, see Creating Collations on page 5-55.

Altering Comments

The COMMENT statement can add a comment to existing comments for an object or replace existing comments with a new one.

You can use the CLEAR clause to clear the existing comments and add a new comment to the object. If you do not use the CLEAR clause, the comment is added as a new row in the COMMENTS catalog table after any existing comments for the object.

Dropping Objects From a Database

The active data dictionary provides the mechanism to drop objects easily from the database as the application requirements change. When an object other than a collation is dropped, SQL/MP ensures the integrity of the database by dropping or invalidating associated dependent objects. This effect on other objects must be carefully reviewed before any object is dropped.

To ensure that collations are not dropped when other objects or SQL programs still need to use them, SQL does not drop a collation that has any dependent objects or programs.

SQL provides the DROP statement and the PURGE utility to delete objects from the data dictionary. The DROP statement operates on a specified object, deletes the catalog definitions, and purges the physical file, if any. The PURGE utility provides the same capability, but the utility enables you to identify objects with file-set lists.

To logically remove specific objects from your database, use the DROP statement; you must use DROP to remove catalogs or constraints. If you need to remove groups of objects as file sets, however, using the PURGE utility is the fastest way.

Note. To delete an SQL program stored in an OSS file, use the appropriate OSS utility to delete the pathname. The file is purged when the last link to the file is removed. For more information, see the *Open System Services Shell and Utilities Reference Manual*.

Both DROP and PURGE require authority to purge the object and any dependent objects. Write authority is required for the catalogs in which the objects are described. For details on authorization requirements, see the *SQL/MP Reference Manual*.

This table summarizes the objects you can remove from the database and the statements and commands that perform the operations. All these operations are discussed in the following paragraphs except for dropping damaged SQL objects, described in Purging Damaged Objects With the CLEANUP Utility on page 11-29.

Object	Operation	Statement or Command
Catalog (all tables)	Delete	DROP CATALOG
Table	File and definition	DROP TABLE PURGE
	Data	PURGEDATA
View	File label and definition	DROP VIEW PURGE
Index	File and definition	DROP INDEX PURGE
Partition	Drop	ALTER TABLE DROP PARTITION ALTER INDEX DROP PARTITION
Column	Delete	DROP TABLE
Constraint	Definition	DROP CONSTRAINT
Collation	Drop object	DROP COLLATION PURGE
Comment	Definition	COMMENT
Damaged SQL object	Drop	CLEANUP

For information about programs, see Section 10, Managing Database Applications.

Dropping Catalogs

To drop a catalog, use the DROP CATALOG statement. The catalog must be empty of all user-defined SQL objects but will still contain the catalog tables and catalog table index definitions.

To drop a catalog, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Query the catalog tables for objects described in this catalog. Use the DISPLAY USE OF command on each object to determine the dependencies.
3. Drop all the objects from the catalog: each user-defined table, view, index, constraint, collation, comment, partition, and program registered in the catalog.
4. Enter the DROP CATALOG statement.

Dropping Tables

To drop a table, use the DROP TABLE statement. Dropping a base table with dependencies is essentially dropping each of the dependent objects separately. SQL drops all the dependencies automatically. These guidelines apply:

- To have the authority to drop a table, you must have all the security and authority required to drop or invalidate all dependent objects, including access to all the catalogs describing all the dependent objects.
- When you drop a table, the operation invalidates the programs that depend on that table. Dropping a table can be very complicated if the table has many dependent objects.
- Dropping a table also drops the table definition and the definitions of all dependent indexes, views, constraints, and comments from the data dictionary. To re-create the environment, you must recover these definitions from backup tapes or OBEY recovery files.
- Dropping a table does not drop any collations used by the table columns.

To drop a table, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine all the dependent objects of the table by using the DISPLAY USE OF command.
3. Prevent all access to the table and its dependent objects.
4. Enter the DROP TABLE statement.

If you plan to use the TMF subsystem for recovering an audited SQL table, see Recovering Purged SQL Tables on page 11-14 before proceeding.

Dropping Views

Dropping a view with the DROP VIEW statement is similar to dropping a table, because the operation drops all the dependent views and invalidates all programs that use the view. A view, however, contains no physical data.

Dropping a view with dependencies is essentially the same as dropping each of the dependent objects separately. SQL drops all the dependent objects automatically, but not dependent programs. These guidelines apply:

- The DROP VIEW statement does not affect any underlying tables; indexes, constraints, or programs that use the underlying tables; or collations used by view or table columns. Likewise, views that are not dependent on the view you are dropping are not affected.

- Dropping the view also drops the view definition from the data dictionary. To re-create the environment, you must recover these definitions from backup tapes or OBEY recovery files.

To drop a view, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine all the objects and programs dependent on the view by using the DISPLAY USE OF command.
3. Prevent all access to the view and its dependent objects.
4. Stop execution of any dependent programs.
5. Enter the DROP VIEW statement to the SQLCI session.
6. For an audited view, make a new TMF online dump so that file recovery does not replace the view.
7. If you want to use the dependent programs again, revise program source files to delete references to the dropped view and recompile the programs.

Dropping Indexes

To drop an index, use the DROP INDEX statement. This statement purges the physical file that contains the index and eliminates the access path to the underlying table.

Dropping an index invalidates programs that depend on the underlying table. You should include steps to explicitly SQL compile the dependent programs to avoid automatic recompilation and to return the application to a valid state.

To drop an index, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the table for which you want to drop the index.
3. Determine which programs depend on the table by using the DISPLAY USE OF command. These programs will be invalidated.
4. Optionally, prevent the use of the table for the duration of the DROP INDEX operation to eliminate conflicts in access to the table; this operation requires exclusive use of the table.
5. Enter the DROP INDEX statement.
6. SQL compile the invalidated programs.
7. Restart use of the table if you stopped its use.

If you plan to use the TMF subsystem for recovering an audited SQL index, see Recovering Purged SQL Tables on page 11-14 before proceeding.

Dropping Partitions of Tables and Indexes

Use the ALTER TABLE statement with the DROP PARTITION option to drop a partition of a key-sequenced, entry-sequenced, or relative table and the ALTER INDEX statement with the DROP PARTITION option to drop a partition of an index.

Determining When to Drop a Partition

When all information in a partition becomes obsolete, or when a database design deficiency leaves a partition continually empty, a reference to a table or index defined across this partition results in an unnecessary message being issued to the partition.

For example, an index label is updated to include the names of all index partitions whenever the label for an associated object is altered. This update can happen, for example, when a table is backed up or restored or when an index is added or dropped. (Simply recompiling a program does not update the labels for the referenced objects.)

This unnecessary message results in a correspondingly longer access time to the table or index. In such circumstances, you might want to drop this partition while leaving the others defined for the object intact.

Guidelines for Dropping Partitions

You can drop partitions of tables and indexes within these guidelines:

- The partition must be empty.
- The partition cannot be the primary partition of the table or index.
- For a relative or entry-sequenced table, you can drop only the last partition of that table.
- All partitions of the table or index must be available when you enter the ALTER statement with the DROP PARTITION option.
- Dropping a partition of a table also drops the corresponding partition of any protection views defined on the table.
- Dropping a partition of a table invalidates all programs that use the table or a view that depends on the table unless a program was compiled with the CHECK INOPERABLE PLANS option and the similarity check is enabled for the table and any associated protection views. (Similarity checking is not available for shorthand views.) For more information, see Using Similarity Checks on page 10-15.
- Dropping a partition of an index invalidates all programs that use the underlying table or a view that depends on that table unless a program was compiled with the CHECK INOPERABLE PLANS option and the similarity check is enabled for the table and any associated protection views. (Similarity checking is not available for shorthand views.)
- You should include steps to explicitly SQL compile dependent programs to avoid automatic recompilation and to return the application to a valid state.

Steps for Dropping Partitions

This example drops an empty partition of a key-sequenced table:

```
>>  ALTER TABLE $VOL1.SALES.CUSTOMER
+>     DROP PARTITION $VOL5.SALES.CUSTOMER;
--- SQL operation complete.
```

To drop a partition, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the table or index from which you want to drop the partition.
3. Determine if the partition is empty by using the FILEINFO command to check the end-of-file indicator for the partition.
4. Determine which programs depend on the table by using the DISPLAY USE OF command. These programs will be invalidated.
5. Enter the ALTER TABLE or ALTER INDEX statement with the DROP PARTITION specification.
6. SQL compile the invalidated programs.

If you plan to use the TMF subsystem for recovering an audited SQL table or index, Recovering Purged SQL Tables on page 11-14 before proceeding.

Deleting Columns

Deleting columns from a table is not allowed. If you want to prevent access to a column of a table, you might create a protection view of the table, excluding the column you want to drop. This method does not physically alter the table structure but essentially masks the unwanted column. This method can work only if the excluded column is defined with a default value. If the excluded column is defined with the NO DEFAULT clause, no user can perform update or insert operations through the view.

To physically delete a column, you must create a new table as follows:

1. Rename the old table and create a new table definition, excluding the columns you do not want in the new table.
2. After creating the new table, load the old table's data into the new table with the LOAD or COPY command, eliminating the missing columns.
3. After the LOAD or COPY operation completes, drop the old table.

You cannot drop columns from views or indexes. To remove a column from a view or index, you must drop the existing object and create a new object, excluding any unwanted columns.

For more information, see Altering Columns on page 7-26.

Dropping Constraints

Dropping constraints on the database is similar to making a program change. Any future data inserts or updates will not have to satisfy the constraint. The DROP CONSTRAINT statement drops only the constraint definition from the catalog and does not affect the data in the table.

Dropping a constraint on a table invalidates the programs that depend on the table. You should include steps to explicitly SQL compile the dependent programs to avoid automatic recompilation and to return the application to a valid state.

To drop a constraint, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the table for which you want to drop the constraint.
3. Determine which programs depend on the table by using the DISPLAY USE OF command. These programs will be invalidated.
4. Optionally, prevent the use of the table for the duration of the DROP CONSTRAINT operation to eliminate conflicts in access to the table; this operation requires exclusive use of the table.
5. Enter the DROP CONSTRAINT statement.
6. SQL compile the invalidated programs identified by the DISPLAY USE OF command in Step 3.
7. Restart use of the table if you stopped its use.

Dropping Collations

To drop a collation, use the DROP COLLATION statement. This statement drops the collation only if no objects or programs depend on it.

To drop a collation, you must own the collation and have authority to read and write to the catalog in which the collation is registered. Follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine all the objects and programs dependent on the collation by using the DISPLAY USE OF command.
3. Drop any dependent objects and programs.
4. Enter the DROP COLLATION statement.

Dropping Comments

You can drop comments at any time with no effect on the database. To drop comments, use the COMMENT statement with the CLEAR option. The CLEAR option drops all comments on the specified object.

This example drops all comments on a constraint on the DEPT table:

```
>>  COMMENT ON CONSTRAINT MGRNUM_CONSTRAINT ON DEPT
+>      IS "" CLEAR;
```

Purging SQL Objects and Enscribe Files

Use the PURGE command to delete a set of SQL objects and Enscribe files specified in a qualified file set list. The PURGE command deletes the table or file and the description in the catalog for SQL objects. PURGE also deletes dependent objects.

The results of a PURGE command are very similar to the results of a DROP statement.

If you want to purge only the data from an audited or nonaudited SQL table, use the PURGEDATA command. This command deletes the data in the table and leaves the table itself (the catalog description) intact. For more information about using PURGEDATA, see Purging Data From SQL Tables on page 8-18.

Using DROP or PURGE

This list summarizes the differences between the operations of the DROP statement and PURGE command:

- PURGE allows many objects identified by a qualified file set list to be purged with one command. The DROP statement drops one object at a time.
- PURGE allows a file set list that contains both SQL objects and Enscribe files. DROP operates only on SQL objects.
- PURGE includes the ALLOWERRORS clause. If ALLOWERRORS is ON, the command tries to purge the specified file set list regardless of the number of errors that are encountered. If ALLOWERRORS is OFF or if you are using the DROP statement, the first error encountered terminates the statement.
- PURGE includes the LISTALL clause. If you specify LISTALL, you receive a confirming message for each purged object. If you use the DROP statement or if you omit the LISTALL option from the PURGE command, you receive no confirming message about each purged object.
- PURGE enables you to browse a selected file set list and select the objects to be purged.
- Both PURGE and DROP require the same security and authorization to purge or drop objects.

- Both DROP and PURGE automatically initiate a TMF transaction for the operation if one has not already been started. You cannot drop or purge a nonaudited object within a user-defined transaction.
- Only the DROP statement is valid for purging a constraint or catalog.
- PURGE enables you to purge shadow labels, as described under Managing Shadow Disk Labels on page 11-36.
- Both the DROP statement and PURGE command can be entered interactively through SQLCI. Only DROP, however, can be used programmatically.

Renaming Objects

You can rename most objects by using an ALTER statement with the RENAME option. This table lists the objects you can rename and the ALTER statements for renaming.

Object	**Operation**	**Statement**
Table	Rename	ALTER TABLE
View	Rename	ALTER VIEW
Index	Rename	ALTER INDEX
Collation	Rename	ALTER COLLATION
Constraint	(Not Allowed)	(Not Applicable)

To rename these objects, use these guidelines:

- You can rename an object only on the same volume where it already exists.
- When you rename a table, all associated protection views, shorthand views, and indexes must be accessible during the renaming so that catalog references can be altered.
- When you rename an index, the underlying table must also be accessible so that catalog references can be altered.
- When you rename a protection view, the underlying table and indexes must be accessible so that file-label references can be altered.
- When you rename a shorthand view, the underlying protection view and table must be accessible so that catalog references can be altered.
- When you rename a partitioned table or index, all partitions of that object must be available. All partitions are automatically renamed.
- When you rename a collation, all dependent objects must be available so that file-label and catalog references can be altered.

These are examples of requests for rename options:

```
>>  ALTER TABLE SALES.CUSTOMER RENAME NYSALES.CUSTOMER;
--- SQL operation complete.
>>  ALTER INDEX SALES.CUSTNAME RENAME NYSALES.CUSTNAME;
--- SQL operation complete.
```

To rename an object, follow these steps:

1. Start an SQLCI session. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
2. Determine the name of the object that you want to rename.
3. Make sure that the new name is not already in use by entering the FILEINFO command.
4. Enter the ALTER TABLE, ALTER INDEX, ALTER VIEW, or ALTER COLLATION statement with the RENAME specification for the appropriate object.
5. SQL compile the invalidated programs.
6. For an audited table, index, or view, make a new TMF online dump.

8 Reorganizing Tables and Maintaining Data

Sometimes you might need to restructure the data in a table by reloading or reorganizing the table. You might do this restructuring, for example, when you want to perform operations such as:

- Reload the data to increase the data-block free space, reducing block splits during insertions and updates
- Partition or repartition a table or an index
- Split a partition to create additional space by distributing data across volumes
- Organize data blocks to eliminate empty space and reduce the number of index levels
- Increase the available disk space when the data or index block structure becomes very fragmented
- Reduce empty but allocated free space that occurs when a large number of records are deleted from a file
- Compress a file to improve disk space usage

An essential step in creating a DSS database is to populate the tables with data derived from operational systems. SQL/MP provides the SQLCI LOAD utility to move data into a table.

To maintain the data in a DSS database, you must periodically add new data and purge the oldest data from the database. SQL/MP provides several SQLCI utilities, including APPEND, COPY, and PURGEDATA, to perform these data-maintenance operations.

Choosing a Reorganization Method

Three methods are available to perform the restructuring function: online reorganization through the RELOAD command of FUP, offline reorganization through the LOAD or COPY command of SQLCI, and physical reorganization through partition move operations.

These restructuring methods have these capabilities and restrictions:

- Reorganizing a table online with the FUP RELOAD command:
  - The table must be a key-sequenced table.
  - The table is accessible for use by the application at all times because of the shared-mode reorganization method.

- The table is reorganized within the physical or partition structure that already exists; with RELOAD, data is reorganized within the current structure only.
- The table cannot use index or data compression.

- Reorganizing a table offline with the SQLCI LOAD or COPY command:
  - If the table is relative or entry-sequenced, this second method must be used. If the table is key-sequenced, the offline method can be used as an alternative to the online reorganization method.
  - If you use the LOAD command, the table must not be audited by the TMF subsystem. If you use COPY, the table can be audited.
  - If the object is a relative or entry-sequenced table, it must be repartitioned.
  - If the table has a new physical column layout, caused by adding or dropping a column in the middle of the table, this reorganization method must be used. The new table layout must be created; then, the data from the old table must be loaded or copied into the new table.

  For more information about using LOAD and COPY, see Loading, Copying, Appending, and Purging Data on page 8-7.

- Reorganizing values stored in a multipartitioned key-sequenced table or index by moving or splitting partitions or by redefining row boundaries:
  - An existing partition of a key-sequenced table or index can be moved, split, or merged with another partition by using the ALTER TABLE (or ALTER INDEX) PARTONLY MOVE statement, as explained under Splitting, Moving, and Merging Partitions on page 7-20.
  - Empty partitions can be added to an SQL table.

Reorganizing a Database Online

You can reorganize only key-sequenced, audited tables online. If the table you need to reorganize is not of this type, you must use the offline approach described under Loading, Copying, Appending, and Purging Data on page 8-7.

Reorganizing Key-Sequenced Files

The RELOAD command of FUP lets you reorganize a key-sequenced file while the file remains available for use by the application. The RELOAD operation physically restructures the file to improve access performance and space usage. The result of RELOAD is equivalent to using a LOAD command to load the source file to a target file of the same format. The RELOAD command, however, allows for shared read-write access to the file during the operation.

Before performing a RELOAD operation, consider these issues:

- The file must be key-sequenced.
- The operation can cause degraded performance. You can, however, control the amount of degradation by using the command's RATE option. The higher the rate, the faster the reload occurs, but the more performance degrades. Conversely, the lower the rate, the slower the reload occurs, but the less performance degrades. The default value for RATE is 100 percent.
- For tables audited by the TMF subsystem, the operation generates audit-trail records describing the movement of data within the file. The total amount of audit-trail data generated for any given file cannot be calculated exactly. For a large file with a lot of data movement, however, the amount can be two to three times the total number of rows in the table. For this reason, the parameters governing the use of TMF audit trails might need to be increased to accommodate the audit-trail data. Or, as a more convenient alternative to increasing these parameters, you might want to increase the frequency of the audit-trail dumps. For more information, see the *TMF Operations and Recovery Guide*.
- The RELOAD command reorganizes either a table or an index independent of each other.
- Three RELOAD parameters control block slack:
  - DSLACK controls the amount of free space in a table's data blocks. The default value for this parameter is 15 percent.
  - ISLACK controls the amount of free space in a table's index blocks. The default value for this parameter is 15 percent.
  - SLACK controls the amount of free space in both index and data blocks. The default value for this parameter is 15 percent.
- When the RELOAD command is issued, FUP initiates a background process to perform the operations requested by the command. After the process is initiated, FUP displays the message RELOAD STARTED and either returns a prompt or terminates (depending upon whether FUP was initiated interactively or noninteractively).
- The RELOAD operation might take a long time, depending upon the size of the file and the rate specified for the command.
- You can suspend the RELOAD operation or request a status report about the progress of the operation, as explained in the next subsection, Determining the Status of a Reorganization.

This command initiates a RELOAD operation for the table named CUSTOMER. The SLACK option sets a minimum amount of free space in the blocks.

```
13> FUP
   - RELOAD $VOL.SALES.CUSTOMER, RATE 30, SLACK 50
```

Determining the Status of a Reorganization

The FUP STATUS command reports the status of a RELOAD operation. Use the STATUS command to determine if the operation has completed or has been suspended and to find out what percentage of the file has been processed.

This command requests the status of the RELOAD operation started in the previous example for the table CUSTOMER:

```
14> FUP
   - STATUS $VOL.SALES.CUSTOMER
```

The status is reported in this format:

```
    OPERATION INITIATED date-time of initiation
    DSLACK = 50%
    ISLACK = 50%
    RATE   = 30%
    10% COMPLETED
```

If the operation is completed, terminated abnormally, or suspended, FUP displays an appropriate message.

Suspending a Reorganization Operation

During the reorganization of a table, you might need to suspend the reorganization process. In most cases, the suspension is necessary for performance considerations. Later, you can restart the reorganization, causing the process to continue from the point where it left off.

This command suspends the RELOAD operation started in the previous example for the table CUSTOMER:

```
15> FUP
   - SUSPEND $VOL.SALES.CUSTOMER
```

If you issue a STATUS command for the RELOAD operation suspended for the CUSTOMER table in the preceding example, the status message follows that command:

```
16> FUP
   - STATUS $VOL.SALES.CUSTOMER
    OPERATION INITIATED date-time of initiation
    OPERATION SUSPENDED date-time of suspension
    DSLACK = 50%
    ISLACK = 50%
    RATE   = 30%
    10% COMPLETED
```

When you want to restart the reorganization of the table, reissue the RELOAD command. In this case, you must omit the NEW option so that FUP restarts a previously suspended reload operation.

If you want to change the RATE or SLACK option, you can do so in the restart command. This command restarts the RELOAD operation for the table CUSTOMER, but adjusts the reloading rate to 20 percent:

```
17> FUP
   - RELOAD $VOL.SALES.CUSTOMER, RATE 20, SLACK 50
```

If you want to keep the same RATE and SLACK values when you restart the reorganization process, enter the RELOAD command without the RATE and SLACK parameters:

```
18> FUP
   - RELOAD $VOL.SALES.CUSTOMER
```

After a RELOAD operation has been suspended and you want to start this operation completely over again, enter the RELOAD command with the NEW option:

```
19> FUP
   - RELOAD $VOL.SALES.CUSTOMER, NEW, RATE 30, SLACK 50
```

In the RELOAD command, the NEW option is necessary only when restarting a RELOAD operation over again from the beginning, following a RELOAD suspension.

Reorganizing Partitions

As table partitions become full, you can reorganize the structure of a table or index by adding, splitting, or dropping a partition, redefining the row boundaries of a partition, changing file extent values, adding empty partitions, or creating and loading a new table.

For many of these operations, you can specify the WITH SHARED ACCESS option to retain full read and write access to data throughout most of the operation. Thus, many of these operations can be performed almost entirely online. (Some operations require the WITH SHARED ACCESS option.)

Before requesting these operations, carefully examine your situation and the desired effect of the operation. For specific information about each of these operations, see Section 7, Adding, Altering, Removing, and Renaming Database Objects.

Balancing Partition Sizes

When a partition of a table or index becomes full, you can split the partition to make room for additional insert and update operations. Suppose, for example, that a table has three partitions based on the CUSTNUM key. The table is partitioned according to the customer number (CUSTNUM) ranges 1 through 2999, 3000 through 5999, and 6000 through 99999, respectively.

Eventually, the last partition becomes full because most new customers are assigned a customer number in the higher ranges of the table (over 6000), adding records to the last partition at a higher rate than the other two stable partitions. In this case, you should split the last partition into two new partitions, one with customer numbers 6000 through 9999 and the other with numbers 10000 through 99999, respectively. When the last partition becomes full again at a later time, this operation can be repeated.

Consider again the case of the table with three partitions based on the CUSTNUM key. The partitions are defined by the customer number (CUSTNUM) ranges 1 through 2999, 3000 through 5999, and 6000 through 99999, respectively. As time passes, the partitions fill 30, 75, and 100 percent of the available space, respectively. You are still limited, however, to only three disk volumes on which you can place partitions.

Ideally, you want to have three partitions that are each about 60 percent full. To achieve this goal, you can reorganize the table by moving the partition boundaries. The FIRST KEY values of the three partitions might now need to indicate the ranges 1 through 4999, 5000 through 7999, and 8000 through 99999, respectively. To perform this operation, you can use the PARTONLY MOVE option of the ALTER TABLE statement.

For more information about splitting partitions and moving partition boundaries, see Splitting, Moving, and Merging Partitions on page 7-20.

If a partition remains empty due to a design or data miscalculation, you can drop the partition. For key-sequenced tables and indexes, you can drop any empty partition—even one that lies in the middle of a set of partitions. The one exception is that you cannot drop the primary partition of a table, although it is empty. For relative and entry-sequenced tables, however, you can drop only the last partition of the table. For more information, see Dropping Partitions of Tables and Indexes on page 7-32.

Changing Extent Size Values

Partitioning might not always be required. For instance, if an entry-sequenced table caused an error 45 (file is full), the error might be based only on the EXTENT SIZE and MAXEXTENTS values specified when the file was created. If the file is full and does not need to be spread across disk volumes, you can increase the MAXEXTENTS value by using the SQLCI ALTER TABLE statement:

```
>> ALTER TABLE PARTS MAXEXTENTS nnn;
```

In this statement, *nnn* is a number greater than the current MAXEXTENTS value. The maximum value allowed for MAXEXTENTS is 959 for primary partitions and 940 for secondary partitions.

The same guideline applies to a particular partition of a table; that is, you can increase the MAXEXTENTS value for a single partition to allow for additional growth.

Adding Empty Partitions

If a table is filling its primary partition or its last secondary partition, you can add an empty partition after the last existing partition to allow for growth. For relative or entry-sequenced tables, the partition is always added to the end of the table. This type of extension spreads the access to the table over additional volumes or nodes. For a table of any organization, an empty partition can be added at any time. If any partition of a key-sequenced table or index is nearly full, split the partition by using an ALTER TABLE or ALTER INDEX statement, as described in Altering Database Objects on page 7-13.

Loading, Copying, Appending, and Purging Data

SQL/MP provides four utilities to move data into or out of tables and Enscribe files offline: the SQLCI CONVERT, LOAD, APPEND, and COPY utilities. For LOAD, APPEND, or COPY, the source file can reside on either disk or tape.

The CONVERT utility uses the LOAD utility to move the data from Enscribe files to SQL tables. If you are converting Enscribe data files into SQL tables, use the CONVERT utility directly.

The LOAD and COPY utilities have very similar options and perform similar tasks. The basic differences follow:

- LOAD enters data into an empty target table. The COPY utility appends or inserts rows into an existing table without purging existing data.
- LOAD provides options for processing key-sequenced files. You can specify, for instance, that the rows are in sorted order. You can specify a maximum number of rows to be loaded. You can define the DSLACK, ISLACK, and SLACK percentages allowed. LOAD reorders unsorted input data and optionally uses a user-specified scratch file location for the SORTPROG processes.
- The LOAD utility is faster than COPY. LOAD sorts rows and then writes them in blocks to the target table. COPY does multiple inserts, one row at a time.
- LOAD enters data into any indexes already created on the table, overwriting any existing data. COPY automatically inserts rows into indexes that already exist.
- LOAD provides an option to load individual partitions of a table.
- You can use COPY to copy data within a user-defined TMF transaction on an audited table. You cannot use LOAD to load an entire audited table without resetting the AUDIT attribute, but you can use LOAD to load data into a single partition of an audited table.
- You can use COPY to copy data from Enscribe unstructured files; LOAD does not operate on unstructured files.

- The COPY utility has a DISPLAY FORMAT option that enables you to display data on your terminal or printer.
- COPY can write to a tape file.

The APPEND utility appends data to an existing table or to a partition of a key-sequenced table. The APPEND utility is like the LOAD utility except that it adds data to a table without purging the existing data. The LOAD and APPEND utilities have similar options and rules. For more information about APPEND, see Appending Data to Tables or Partitions on page 8-15.

Guidelines for Loading Tables

When you load tables, consider these guidelines:

- You cannot use the LOAD command to load an entire audited table. If the table is audited, you must alter the AUDIT attribute to NO AUDIT before performing the load. After completing the load, you must alter the AUDIT attribute back to AUDIT and then make a TMF online dump of the table.

  When loading a single partition, you need not alter the AUDIT attribute. You must, however, do an online dump of the partition when finished with the load to preserve TMF recovery capability. For more information about loading a single partition, see Loading Individual Partitions on page 8-9.
- Each source record of an Enscribe file is written as a row to the target table (or each row of the source table is written as a target record of an Enscribe file), overwriting any existing data. The operation must satisfy constraints and provide a corresponding source column value for all columns defined with NO DEFAULT.
- If you are loading a table that uses a SYSKEY primary key or a clustering key with the SYSKEY column appended to the clustering key, the values of the SYSKEY column will change. If your application has used SYSKEY values, alone or with a clustering key, as a user-defined embedded linkage field in other tables, a reloading of the table invalidates your linkage of the tables. Use of a SYSKEY column, alone or with a clustering key, precludes this reload operation.
- A relative table uses a relative record pointer. If you load a relative table, the record automatically compacts, deleting unused slots. To avoid this outcome, use the NO COMPACT option to retain the relative record locations.
- Three LOAD parameters control block slack:
  - DSLACK controls the amount of free space in a table's data blocks.
  - ISLACK controls the amount of free space in a table's index blocks.
  - SLACK controls the amount of free space in both data and index blocks.
- LOAD automatically loads any indexes defined for the target table but does not automatically load any alternate-key files associated with a target Enscribe file.

- If the LOAD utility fails during the LOAD operation, the target table or file is left in an invalid state and is unusable. You can restart the LOAD operation to overwrite the existing data using the same source file.
- If you are loading data from an Enscribe file into an SQL table, from an SQL table into an Enscribe file, or from an SQL table into another SQL table, the data types of the source and target fields or columns must be compatible. The rules for valid field or column compatibility are the same as those described for the CONVERT utility in the *SQL/MP Reference Manual*.
- You must create all target files or tables before issuing the LOAD command. LOAD does not create the files or tables.

Loading Individual Partitions

The LOAD utility allows partitions of tables to be loaded separately. You must perform these operations carefully to ensure that all partitions are loaded with logically consistent data.

You can use the DataLoader/MP product to load data into multiple partitions of a fact table or history table. Although you can use DataLoader/MP to load data into any SQL/MP table, it is primarily useful for loading and maintaining large tables such as those used in a data warehouse. With DataLoader/MP, you can initially populate a data warehouse with data derived from an operational database; you can also perform periodic load operations that update an existing data warehouse. DataLoader/MP can use the LOAD utility to perform load operations. Therefore, users of DataLoader/MP, in addition to users of the LOAD utility, should read the following description of loading partitions. For more information about the DataLoader/MP product, see the *DataLoader/MP Reference Manual*.

Loading a Single Partition

This example loads a secondary partition of the ORDERS table that resides on $VOL1.MKT when the primary partition resides on $VOL4.MKT:

```
>> LOAD $OLD.SALES.ORDERS, $VOL1.MKT.ORDERS, PARTONLY;
```

When you load a single partition of an audited table, you need not reset the AUDIT attribute before the load operation. This action is only required when loading an entire audited table. Be sure to do an online dump of the partition when the load is finished if you want to preserve TMF recovery capability.

Loading Multiple Partitions in Parallel

The PARTONLY option lets you load partitioned base tables in parallel. This strategy can improve load performance if table partitions are distributed across disks, processors, and I/O channels. These steps describe possible strategies for loading partitions in parallel:

1. Start an SQLCI process for each partition. One way to do this is to start each SQLCI process in the processor associated with the partition to be loaded. Another way is to start SQLCI processes in the processors associated with the data sources for the LOAD command.
2. Issue one LOAD...PARTONLY command for each SQLCI process (and thus each corresponding partition).
3. Supply each LOAD command with the specific range of input data for the partition it is loading. Three possible strategies are:
   - Arrange the input data such that it is divided into separate files, each containing input for a specific target partition. Use these files as input to the LOAD commands.
   - Do a SORTED load and specify FIRST KEY. When the SORTED option is specified, LOAD stops processing input as soon as it encounters a row beyond the end of the target partition.
   - Use processes to read input data. Start each data source process as a named process before entering the LOAD command. Use the process name as the input file for the LOAD command. The process must wait for requests on its $RECEIVE file and then supply data by replying to those requests. When using this approach, be sure to balance processing for optimal performance.

   You can use the DataLoader/MP product to help implement the preceding tasks. For example, you can use DataLoader/MP to arrange to have the input data delivered to the correct target partitions. For more information about the DataLoader/MP product, see the *DataLoader/MP Reference Manual*.

Examples of Loading Tables

This example loads an SQL table from an Enscribe file. The LOAD command moves the fields in order by using the default MOVEBYORDER option. The fields involved in the transfer from the source file must be compatible with the data type and order of the receiving columns in the target table.

```
>> LOAD $ENSC.SALES.ORDERS, $VOL1.SALES.ORDERS,
+>    SCRATCH $TEMP.SCRATCH.JUNK
+>    SOURCEDICT $ENSC.SALES  SOURCEREC ORDERREC;
```

The next example loads data from one table into another table. You might perform this move to increase lengths of existing columns or to drop columns. Columns are matched by name as specified by the MOVEBYNAME option. The source table must contain a matching column for each column defined in the target table. The columns must have compatible data types, but can be of different sizes.

```
>> LOAD \SYS1.$OLD.SALES.ORDERS, \SYS1.$VOL1.SALES.ORDERS,
+>    SORTED
+>    TRUNCATION ON
+>    MOVEBYNAME;
```

Loading Data From an Enscribe File

This example loads data into a table from an Enscribe file. The LOAD command includes the MOVE option to match fields to columns because DELIV_DATE is missing from the source file and the field names are not the same as the column names. The layout of the Enscribe file $ENSC.SALES.ORDERS precedes the commands that create and load the new table, $VOL1.SALES.ORDERS:

```
* Record Layout for $ENSC.SALES.ORDERS
 01 ORDERS.
    02 ORDER-NUM           PIC 9(6).
    02 ORDERED-DATE        PIC S9(6) COMP.
    02 SALESMAN            PIC 9(4).
    02 CUSTOMER-NUMBER     PIC 9(4).
>> CREATE TABLE $VOL1.SALES.ORDERS
+>    (ORDERNUM           PIC 9(6) NO DEFAULT NOT NULL,
+>     ORDER_DATE         PIC S9(6) COMP DEFAULT SYSTEM NOT
NULL,
+>     DELIV_DATE         PIC S9(6) COMP DEFAULT SYSTEM NOT
NULL,
+>     SALESREP           PIC 9(4) DEFAULT SYSTEM,
+>     CUSTNUM            PIC 9(4) DEFAULT SYSTEM NOT NULL,
+>    PRIMARY KEY (ORDERNUM))
+>    CATALOG $VOL1.SALES
+>    EXTENT(1000,100);
--- SQL operation complete.
>> LOAD $ENSC.SALES.ORDERS, $VOL1.SALES.ORDERS,
+>    SCRATCH $TEMP.SCRATCH.JUNK
+>    SLACK 50
+>    SOURCEDICT $ENSC.SALES  SOURCEREC ORDERREC
+>    MOVE (ORDER-NUM TO ORDERNUM, ORDERED-DATE TO ORDER_DATE,
+>         SALESMAN TO SALESREP, CUSTOMER-NUMBER TO CUSTNUM);
```

This example loads data into a table from an Enscribe file. Some of the source numeric fields contain spaces, which are allowed in Enscribe files but are not allowed in SQL tables. The REPLACE SPACES WITH ZEROS option specifies converting numeric decimal fields from blanks to zeros. Because the MOVEBYORDER option is the default, the fields involved in the transfer from the source file must be compatible with the data types and order of the receiving columns in the target table.

```
>> LOAD $ENSC.SALES.ORDERS, $VOL1.SALES.ORDERS,
+>    REPLACE SPACES WITH ZEROES
+>    SOURCEDICT $ENSC.SALES  SOURCEREC ORDERREC;
```

This example loads data into a table from an Enscribe file. The LOAD command with the MOVE option specifies explicit matching of the source fields and target columns. The source file and target table have these differences:

- DELIV_DATE is a new column not in the source file; this field must be defined with a DEFAULT option to perform the LOAD.
- The CUST_PO column size has been increased by five bytes.
- The ITEM-LIST OCCURS clause has been broken into the 10 element fields.

The Enscribe record layout of the source file follows:

```
* Record Layout for $ENSC.SALES.ORDERS
 01 ORDERS.
    02 ORDER-KEY.
       05 ORDER-NUM                PIC 9(6).
       05 ORDERED-DATE             PIC S9(6) COMP.
    02 SALESMAN                    PIC 9(4).
    02 CUSTOMER-INFO.
       05 CUSTOMER-PO-NUM          PIC X(30).
    02 CUSTOMER-NUMBER             PIC 9(4).
    02 ITEM-LIST                   PIC X(4)
            OCCURS 10 TIMES.
```

The SQL table layout of the target table follows. The table must be created before the LOAD operation.

```
>> INVOKE $VOL1.SALES.ORDERS FORMAT COBOL85;

* Record Definition for table \SYS1.$VOL1.SALES.ORDERS
* Definition current at 09:07:21 - 04/12/89
 01 ORDERS.
    02 ORDERNUM                    PIC 9(6).
    02 ORDER-DATE                  PIC S9(6) COMP.
    02 DELIV-DATE                  PIC S9(6) COMP.
    02 SALESREP                    PIC 9(4).
    02 CUST-PO                     PIC X(35).
    02 CUSTNUM                     PIC 9(4).
    02 ITEM-1                      PIC X(4).
    02 ITEM-2                      PIC X(4).
    02 ITEM-3                      PIC X(4).
    02 ITEM-4                      PIC X(4).
    02 ITEM-5                      PIC X(4).
    02 ITEM-6                      PIC X(4).
    02 ITEM-7                      PIC X(4).
    02 ITEM-8                      PIC X(4).
    02 ITEM-9                      PIC X(4).
    02 ITEM-10                     PIC X(4).
```

Next is the LOAD command to load the Enscribe file into the new SQL table. The MOVE option explicitly names all the field-to-column conversions. The source file OCCURS array is subscripted in the MOVE clause to the target data element of the table.

```
>> LOAD $ENSC.SALES.ORDERS,  $VOL1.SALES.ORDERS,
+>    SORTED
+>    MOVE (ORDER-NUM TO ORDERNUM,
+>          ORDERED-DATE TO ORDER_DATE,
+>          SALESMAN TO SALESREP,
+>          CUSTOMER-PO-NUM TO CUST_PO,
+>          CUSTOMER-NUMBER TO CUSTNUM,
+>          ITEM-LIST(1) TO ITEM_1,
+>          ITEM-LIST(2) TO ITEM_2,
+>          ITEM-LIST(3) TO ITEM_3,
+>          ITEM-LIST(4) TO ITEM_4,
+>          ITEM-LIST(5) TO ITEM_5,
+>          ITEM-LIST(6) TO ITEM_6,
+>          ITEM-LIST(7) TO ITEM_7,
+>          ITEM-LIST(8) TO ITEM_8,
+>          ITEM-LIST(9) TO ITEM_9,
+>          ITEM-LIST(10) TO ITEM_10),
+>    SOURCEDICT $DATA1.EORDERS  SOURCEREC ORDERREC;
```

Loading Data Into an Enscribe File

This example loads data into an Enscribe file from an SQL table. The LOAD command must specify the Enscribe dictionary subvolume unless the dictionary resides on the current default subvolume. MOVEBYORDER is the default field-matching protocol, which requires that the fields are compatible with the columns in the physical order (the order of the fields in the DDL record and the order of the columns in the corresponding table description).

```
>> LOAD $VOL1.SALES.ORDERS, $ENSC.SALES.ORDERS,
+>    SCRATCH $TEMP.SCRATCH.JUNK
+>    TARGETDICT $ENSC.SALES  TARGETREC ORDERREC
+>    MOVEBYORDER;
```

Guidelines for Copying Tables

The COPY utility provides another method of loading tables and files with data. When you copy tables, consider these guidelines:

- If the target is a table, the COPY operation is effectively a set of INSERT statements with the STABLE ACCESS option and, except for key-sequenced tables, the APPEND option. The operation must satisfy constraints and provide a corresponding source-column value for each column defined with NO DEFAULT. The data types of the source and target fields or columns must be compatible.
- If the target is an unstructured, relative, or entry-sequenced file, data is appended to the end of the file.
- For relative input files, the COMPACT option controls whether zero-length records are ignored or written.
- COPY automatically copies values to indexes of a target table. COPY also updates alternate-key files of a target Enscribe file.

- You can copy within a user-defined TMF transaction or, if the table is audited, COPY initiates the transaction. If you are copying large amounts of data, you must plan to ensure that the TMF audit trail space is large enough to handle the copied rows.
- For an audited table, you can alter the AUDIT attribute of the table to NO AUDIT before the COPY operation. After the COPY operation completes, alter the attribute to AUDIT again; then make a TMF online dump, because any previous dump of the table is no longer valid.
- If the COPY operation fails or is stopped during the data loading, the state of the target table or file depends on whether it is audited or nonaudited. If the target is audited, the TMF transaction terminates abnormally, and the work is undone by the TMF subsystem. If the target is nonaudited, all the inserted rows are committed.
- The rules for valid field compatibility are the same as those described for the CONVERT utility in the *SQL/MP Reference Manual*.

Examples of Copying Tables and Files

Several examples of copying tables and files follow.

The first example copies data into a table from an Enscribe file. The COPY command maps the fields with MOVEBYORDER and specifies truncation if necessary. The fields involved in the transfer from the source file must be compatible with the data type and order of the receiving columns in the target table.

```
>> COPY $ENSC.SALES.ORDERS, $VOL1.SALES.ORDERS,
+>    MOVEBYORDER ON
+>    TRUNCATION ON
+>    SOURCEDICT $ENSC.SALES  SOURCEREC ORDERREC;
```

The next example copies data from one table into another table, with only those rows in which EMPNUM is greater than or equal to 4000 copied to the target table. The tables have identical definitions.

```
>> COPY \SYS1.$VOL1.PERSNL.EMPLOYEE,
+>        \SYS1.$VOL1.SPECIAL.EMPLOYEE,
+>    FIRST KEY 4000
+>    MOVEBYNAME;
```

This command copies data into a table from a tape file:

```
>> COPY $TAPE, $VOL1.SALES.CUSTOMERS;
```

This example copies data from a table to a terminal in hexadecimal format:

```
>> COPY $VOL1.SALES.CUSTOMERS, \SYS1.$TERM1,
+>    HEX;
```

This example copies 100 rows from a table to a terminal in ASCII format:

```
>> COPY $VOL1.SALES.CUSTOMERS, \SYS1.$TERM1,
+>    ASCII COUNT 100;
```

Appending Data to Tables or Partitions

The APPEND utility adds data to the end of a table or partition of a key-sequenced table. The APPEND utility is a form of the LOAD utility adapted for a specific purpose; it performs at the same speed as LOAD. Thus, the APPEND and LOAD utilities have similar options and perform similar tasks.

The APPEND utility preserves the existing data in the target table; in this respect APPEND is similar to COPY and differs from LOAD. For a comparison of the LOAD and COPY utilities, see Loading, Copying, Appending, and Purging Data on page 8-7.

The APPEND utility is especially useful in a DSS environment. To keep a data warehouse up to date, you can use APPEND to perform periodic (for example, daily, weekly, or monthly) updates to the database.

You can use the APPEND utility to append data to multiple partitions of a table; the DataLoader/MP product can help you streamline this task. DataLoader/MP is a nonprivileged batch program that provides a library of utility routines for loading and maintaining SQL tables. DataLoader/MP can use the APPEND utility to perform append operations. For more information about the DataLoader/MP product, see the *DataLoader/MP Reference Manual*.

Guidelines for Appending Data to Tables

When you append data to tables or table partitions, consider these guidelines:

- The APPEND utility adds data to the end of a table or partition without purging existing data. APPEND cannot insert data into arbitrary places in the table. (Use COPY to insert rows between existing rows.)
- APPEND adds data to key-sequenced or entry-sequenced SQL tables. You cannot use APPEND to add data to a relative table, Enscribe file, unstructured file, or any file other than an SQL file.
- APPEND can use any source file that LOAD uses, including a SQL table, Enscribe file, unstructured disk file, tape file, a device such as a terminal, or a Guardian process.
- Like LOAD, the APPEND utility writes rows in blocks to the target table. APPEND is faster than COPY.
- APPEND does not operate on tables with indexes.
- APPEND provides the PARTONLY option to append data to individual partitions of a key-sequenced table. APPEND adds rows with key values logically greater than the last existing row in the target partition. The added key values must be logically less than that of the first row in the next partition.
- APPEND provides options for processing key-sequenced files. You can specify, for instance, that the rows are in sorted order. You can specify a maximum number of rows to be loaded. APPEND reorders unsorted input data and optionally uses a user-specified scratch file location for the SORTPROG processes.

- Three APPEND parameters control block slack:
    - DSLACK controls the amount of free space in a table’s data blocks.
    - ISLACK controls the amount of free space in a table’s index blocks.
    - SLACK controls the amount of free space in both data and index blocks.
- You cannot use APPEND to append data to an entire audited table without resetting the AUDIT attribute, but you can use APPEND to add data to a single partition of an audited, key-sequenced table.
- Applications do not have access to a table being modified by APPEND. If you use the APPEND PARTONLY option to modify a partition, the affected partition is not accessible to applications while APPEND is in progress; other partitions remain available.
- If an error occurs during an append operation, and APPEND is able to terminate gracefully, no new data is added to the target table. To determine if an append operation succeeded, check the SQLCI listing to see if error messages occurred.
- If a processor failure, process failure, BREAK command, or another event interrupts an append operation, and APPEND cannot terminate gracefully, the target table remains inaccessible to applications. You must use the APPENDRESTART or APPENDCANCEL command to restore the target table to its original state and either complete the append operation (with APPENDRESTART) or cancel the operation (with APPENDCANCEL).

  For more information about the APPENDRESTART and APPENDCANCEL commands, see the *SQL/MP Reference Manual*.

For information about loading tables, see Guidelines for Loading Tables on page 8-8.

Appending Data to Multiple Partitions in Parallel

The PARTONLY option lets you append data to partitioned tables in parallel. This strategy can improve append performance if table partitions are distributed across disks, processors, and I/O channels. These steps describe possible strategies for appending data to partitions in parallel:

1. Start an SQLCI process for each partition. One way to do this is to start each SQLCI process in the processor associated with the target partition. Another way is to start SQLCI processes in the processors associated with the data sources for the APPEND command.
2. Issue one APPEND...PARTONLY request for each SQLCI process (and thus each corresponding partition).

3. Supply each APPEND command with the specific range of input data for the target partition. Three possible strategies are:

   - Arrange the input data so that it is divided into separate files, each containing input for a specific target partition. Use these files as input to the APPEND commands.
   - Do a SORTED append and specify FIRST KEY. When the SORTED and PARTONLY options are specified, APPEND stops processing input as soon as it encounters a row beyond the end of the target partition.
   - Use processes to read input data. Start each data source process as a named process before entering the APPEND command. Use the process name as the input file for the APPEND command. The process must wait for requests on its $RECEIVE file and then supply data by replying to those requests. When using this approach, be sure to balance processing for optimal performance.

   You can use the DataLoader/MP product to help implement the preceding tasks. For example, you can use DataLoader/MP to arrange to have the input data delivered to the correct target partitions. For more information about the DataLoader/MP product, see the *DataLoader/MP Reference Manual*.

Example of Appending Data

This example assumes you have a history table containing 80 weeks of data. You could partition the table so that each week of data resided on one partition. This scheme, however, might create disproportionate requests for data from certain partitions, particularly those containing the most recent weeks of data. Therefore, the table is partitioned by a hash value so that each week of data is striped (partitioned) across 16 partitions. To maximize parallel execution, each partition is associated with a different processor.

The example uses the APPEND utility each week to add the most recent week of data to the appropriate 16 partitions. It starts an SQLCI process for each target partition (in each associated processor). The NOWAIT option allows you to run separate, concurrent SQLCI processes—you can enter the next SQLCI command without having to wait for the last process to finish. Prompts, errors, and other messages are directed to separate output files so that you can distinguish events occurring in each process.

The example divides the input data into 16 Enscribe files; each file’s data is appended to the corresponding target partition. The input files reside on volumes $VOL1 through $VOL4. The target partitions reside on volumes $VOL33 through $VOL48. The Enscribe record layout corresponds exactly to the target table layout; therefore, no move options are needed to convert input fields into target columns.

The SORTED option indicates that the input data has already been sorted; the append operation does not need to perform further sorting. The ALLOWERRORS option ensures that append operation will proceed even if an input record contains data that could not be converted into a target column.

```
>  SQLCI / CPU 0, OUT $S.#LOG33, NOWAIT / &
>& APPEND $VOL1.WEEK25.PART33, $VOL33.TARGET.HIST, &
>&    RECOVERYFILE $VOL2.RECOVER.PART33 &
>&    PARTONLY, SORTED, ALLOWERRORS ON ; &
>& EXIT ;
>  |
>  SQLCI / CPU 1, OUT $S.#LOG34, NOWAIT / &
>& APPEND $VOL1.WEEK25.PART34, $VOL34.TARGET.HIST, &
>&    RECOVERYFILE $VOL2.RECOVER.PART34 &
>&    PARTONLY, SORTED, ALLOWERRORS ON ; &
>& EXIT ;
>  |
>  |
>  |
>  SQLCI / CPU 15, OUT $S.#LOG48, NOWAIT / &
>& APPEND $VOL4.WEEK25.PART48, $VOL48.TARGET.HIST, &
>&    RECOVERYFILE $VOL2.RECOVER.PART48 &
>&    PARTONLY, SORTED, ALLOWERRORS ON ; &
>& EXIT ;
```

The example issues 16 similar APPEND commands, one for each target partition. The append operation executes in parallel against the 16 partitions.

The RECOVERYFILE parameter specifies a file that stores information needed to restore the target partition to its initial state if a process failure or processor failure interrupts the APPEND operation. For more information about this parameter, and about the syntax and use of APPEND options, see the *SQL/MP Reference Manual*.

Purging Data From SQL Tables

If you want to purge only the data from a nonaudited or audited SQL table, use the PURGEDATA command. This command clears only the data, leaving the catalog description of the table valid. These guidelines apply:

- The PURGEDATA operation temporarily invalidates the table and indexes to prevent concurrent access by other users until the data is purged.

  If an error occurs after the table is marked invalid but before the PURGEDATA operation begins, the table is revalidated, and the data remains unchanged. If an error occurs during the PURGEDATA operation and the operation fails to complete, PURGEDATA leaves the table marked as corrupt. To recover, resolve the problem that caused the first attempt to fail, then reissue the PURGEDATA command.

- After purging the data, the PURGEDATA operation validates the table and indexes so that they are again accessible to other users.

- The PURGEDATA operation does not automatically alter the table’s statistics. After purging the data and after you (or any programs) have added data to the table, run an UPDATE STATISTICS statement to record current statistics for the table in the catalog. If the statistics are incorrect, the SQL compiler might not select the best access path for performance.

- You cannot include the PURGEDATA command within a user-defined TMF transaction.
- You cannot use the PURGEDATA command on an SQL program, view, catalog table, or index.
- With the PARTONLY option of the PURGEDATA command, you can purge data from a single partition of a partitioned table without indexes. (If you omit the PARTONLY option but specify a primary or secondary partition in the command, the data is removed from all partitions and indexes.)
- When using the PARTONLY option of the PURGEDATA command for a relative or an entry-sequenced table, you can only purge data from the last partition. For a key-sequenced table, however, you can purge data from any partition. The PARTONLY option applies only to tables with no dependent indexes.

To use PURGEDATA, follow these steps:

1. Start an SQLCI session.
2. Enter a LOG command to initiate a log file for the statements and commands entered in this session. Keep the log for your records.
3. Prevent the use of the table.
4. Enter the PURGEDATA command. To purge data from an individual partition of a table, use the PARTONLY option of this command.

When you perform a PURGEDATA operation on an SQL audited table, the end-of-file marker is moved backward in the table, and the TMF audit record generated contains the before-images and after-images of the altered file label; however, before-images of the data in the table are not generated.

Under these circumstances, no data exists to enable you to roll back the PURGEDATA operation to the previous state. However, if you make periodic online dumps of the table and note the times at which you issue PURGEDATA commands, the purged data is retained in audit images that can be recovered. For this recovery, use one of the TMF interfaces (such as TMFCOM) to issue the RECOVER FILES command with the TIME attribute set for file recovery to the time before the data was purged.

9 Moving a Database

The guidelines required for moving a database depend on knowing the current database scheme used at each site. The database administrator should analyze all the factors and develop a plan before attempting to move a database.

Reasons for Moving a Database

The reason for moving a database usually falls into one of these categories:

- Moving objects to enhance performance

  This category might include moving objects to another volume, splitting or moving partitions across different volumes, or redefining partition row boundaries for tables and indexes.

- Adding or changing equipment

  This category might include moving objects to new volumes, partitioning tables and indexes, or restoring a volume following an equipment change.

- Moving objects from one application development phase or group to another, such as moving the database and application programs from development to production

  The move can be from one subvolume to another, from one volume to another, or from one node to another.

- Moving objects from one node to another, such as the release of software to an end-user node.

  This category includes the creation of new catalogs, the creation or moving of the database, and the SQL compilation of application programs.

- Moving objects from a node running an older version of SQL/MP software to a node running a newer version.

  This category involves a series of staged operations and testing, described in the *SQL/MP Version Management Guide*.

Moving a database involves moving a set of SQL catalogs and objects from one environment to another. For example, if you move a catalog or any of its objects, you probably want to move the objects and all the definitions and relations associated with those objects.

Determining Move Dependencies

The steps you take to move SQL objects or an entire database depend on the layout of your database. The duplication process cannot ignore or overrule the dependency requirements of database objects, described in earlier sections of this manual. The complexity of the layout of your database dictates the complexity of the statements and commands required to move the database.

Before moving a database, consider these issues:

- What are the dependent SQL objects that might be affected by the move?
- Does the moving plan consider the effect of the move on dependent objects or on subsequent moves?
- Does the plan include revalidation of the dependent programs?
- Do you have the authority to move the objects and all the dependent objects?
- Are the underlying tables, partitions, and systems available, as required?
- What media are available and appropriate for the move?
- Do you have sufficient disk space in the target locations?
- How will the move affect the users of the database or the users of the application programs?
- Do you have a valid recovery mechanism for the new database scheme?
- What steps must you take to ensure the consistency of the database during and after the move?
- Are you moving the objects from one SQL/MP release environment to another and planning to upgrade the catalogs?

Choosing Utilities for the Move Operation

To move SQL objects, you must create copies of the objects at the new location. You can create a copy of an SQL table, index, view, or an SQL program stored in a Guardian file, with either the SQLCI DUP utility or the Guardian BACKUP and RESTORE utilities. The SQLCI COPY and LOAD utilities also provide a mechanism for moving SQL tables.

If you only need to rename a table, index, view, collation, or SQL program stored in a Guardian file, you can use an SQLCI ALTER statement with the RENAME option to rename the object. For more information, see Renaming Objects on page 7-36.

For SQL programs stored in an OSS file, use the appropriate OSS utility to move or rename the program. For example, you can use the `mv` utility or the rename() api to rename an OSS file. For more information about OSS utilities, see the *Open System*

Services Shell and Utilities Reference Manual. The remainder of this subsection applies to SQL tables, indexes, views, and SQL programs stored in Guardian files.

△ **Caution.** Some utilities, as well as SQLCI, let you request purge operations on target files that fall within the context of the command issued. When you refer to qualified file-set lists in such a command, the utility might inadvertently purge an object you did not expect to be purged. You can protect your catalogs, tables, and indexes against the effects of an incorrectly specified file set list by using SQLCI to assign the NOPURGEUNTIL attribute to these objects. NOPURGEUNTIL lets you specify an expiration date and time for the objects and prevents them from being removed before that time.

COPY and LOAD

The COPY and LOAD utilities operate only on tables. You must create a table before you can copy or load data into it. The CREATE TABLE LIKE statement creates a new table identical to the original one; this operation simplifies creating a new table. For additional information about using the COPY and LOAD utilities, see Section 8, Reorganizing Tables and Maintaining Data, or the *SQL/MP Reference Manual*.

You can use the COPY command for operations similar to those of the LOAD command. These differences, however, exist between the two operations:

- The COPY command can run within a user-defined TMF transaction, but you must ensure that the audit trails are large enough to contain the data. For LOAD, the table must be nonaudited; therefore, a TMF transaction is not allowed.
- The COPY command does not allow the SLACK option.
- The COPY command can insert data into a table that has data or is empty. The LOAD command removes any data in the target table before inserting data.
- A COPY operation is usually slower than an equivalent LOAD operation.
- COPY can handle two tables whose columns use different collations, while LOAD cannot.

The success of the COPY and LOAD operations depends upon various factors, including the software releases of the source and target SQL/MP systems and the versions of the source and target objects.

A column in a source object can use a different collation than a column in a target object. COPY and LOAD do not consider whether the source object uses collations; this situation can cause duplicate key errors.

DUP and BACKUP/RESTORE

The DUP utility duplicates objects from one location to another interactively. DUP can move SQL objects from one node to another when the nodes are connected in a network and the requesting user ID has the appropriate remote security to allow the transfer.

The BACKUP and RESTORE utilities provide another method of moving SQL/MP objects. You can use these utilities to move objects to another location on the same node or to another node. BACKUP and RESTORE must be used in cases in which the target node is not physically connected to the source node.

When you back up files, you specify a qualified file set to indicate the set of files to be backed up. A qualified file-set list specifies a set of objects and files and optionally includes clauses that restrict the objects and files operated on based on attributes of the objects and files. For information about the syntax of a qualified file-set list, see the "Qualified File-set List" entry in the *SQL/MP Reference Manual* or use the SQLCI HELP command.

This list compares DUP with BACKUP/RESTORE:

- The DUP and BACKUP/RESTORE utilities allow these options:
  - Using qualified file set lists and wild-card characters
  - Working with both SQL/MP objects and Enscribe files
  - Selecting options to keep or purge targets of the file set lists to prevent duplication or overwriting
  - Continuing with an operation despite errors
  - Automatically moving comments and constraints to the target catalog of the target table
- DUP allows these options:
  - Saving current file information with the new file (the SAVEID, SOURCEDATE, and SAVEALL options)
  - Choosing automatic duplication, explicit duplication, or no duplication for protection and shorthand views
  - Choosing either automatic duplication or no duplication for dependent indexes
  - Duplicating the source object only when the command has exclusive use of the object

- BACKUP/RESTORE allows these options:
    - Choosing either automatic duplication or explicit duplication for dependent indexes
    - Using tape handling features
    - Automatically moving protection views with the underlying table
    - Explicitly duplicating shorthand views named in a file set list
    - Automatically SQL-compiling program files upon restoration with the SQLCOMPILE ON option
    - Automatically creating necessary catalogs with the AUTOCREATECATALOG ON option
    - Copying the source object while other users have access to the object
    - Choosing either automatic duplication or explicit duplication of individual table and index partitions

To handle SQL files, both the BACKUP and RESTORE processes must be licensed. During a typical INSTALL operation, both BACKUP and RESTORE are licensed automatically. In special circumstances, sites might deliberately create unlicensed BACKUP and RESTORE object files, but these processes cannot access SQL files.

The success of the DUP and BACKUP/RESTORE operations depends upon several factors, including whether operations involve different versions of SQL/MP software and different versions of SQL objects.

△ **Caution.** If an SQL object has the UNRECLAIMED FREESPACE (F) or INCOMPLETE SQLDDL OPERATION (D) attribute set, do not attempt to back up, move, or duplicate the object until the attribute is reset. For more information, see UNRECLAIMED FREESPACE (F) and INCOMPLETE SQLDDL OPERATION (D) Flags on page 7-24.

Guidelines for DUP Operations

You cannot perform a DUP operation within a user-defined TMF transaction. If the source object is audited, the DUP command completes the DUP operation on a nonaudited target object and then automatically changes the AUDIT attribute to AUDIT as the operation completes. BACKUP and RESTORE can move audited files by using the AUDITED option, in the same manner as DUP. You must make TMF online dumps of all restored audited objects, after using DUP or RESTORE, to create a new recovery point.

Guidelines for Name Mapping in BACKUP and RESTORE

To move interrelated SQL objects from one volume or node to another, you can use the MAP NAMES option. Use of this option requires familiarity with the way in which the BACKUP and RESTORE utilities handle file names. Be sure to specify a target file set list or define the MAP NAMES and CATALOG options correctly for the dependencies of the target object, or the moved objects might be left in an invalid state or might not be moved.

Note. The PARTONLY and MAP NAMES options are mutually exclusive. So, if you use the PARTONLY option during BACKUP and RESTORE, you will not be able to use the MAP NAMES option.

File set Considerations

All files in the specified file set must originate from the same node; you cannot specify two nodes in the BACKUP command.

Although the files in a file set all reside on one node, they might have implicit relationships with files on other nodes. For example, a file might be partitioned across several nodes, or a base table might reside on one node and have indexes on another. Unless you specify otherwise, the BACKUP utility backs up these related files along with the files specified explicitly in the file set.

The BACKUP utility saves the names of the backed-up files in local or network format, depending on the location of the file set relative to the BACKUP process:

- The files are considered “local” if the file set is on the node running the BACKUP utility. Local file names are saved without a node identifier.
- The files are considered “remote” if they reside on a node other than that running the BACKUP utility. That is, if the BACKUP utility is running on node \A and is backing up a file set from remote node \B, the file set defined on remote node \B would be considered “remote” by BACKUP. The BACKUP utility stores the names of files from remote nodes in remote internal format (\<node-number>.<volume>.<subvolume>.<file-identifier>) to prevent accidental overwriting of files that have the same file names.

△ **Caution.** When backing up files, be aware that you cannot currently restore files in a remote file set if their associated node number is not available on the destination network.

This example backs up a remote table and a local index. Suppose that TABLE1 resides on remote node \B and has index TINDX on local node \A. This BACKUP command (issued from node \A) backs up TINDX along with TABLE1:

```
BACKUP $TAPE, \B.$VOL.SUBVOL.TABLE1
```

The files are backed up as:

```
\B.$VOL.SUBVOL.TABLE1          $VOL.SUBVOL.TINDX
```

Restoring Files to Multiple Nodes

When you restore a file set that resides on multiple nodes, the RESTORE utility searches the network for a node that matches the node number of each remote file on the tape.

Whether you are restoring files to the node from which the files were backed up, or to another node on the same network, you need not be concerned about file names or the destination of remote files. Local files are restored on the "local" node (as defined under File set Considerations on page 9-6). Remote files are restored to their original nodes. Thus, if you restore a file set to a node other than their original node, the default behavior results in a node name change for local files but not for remote files. Use the MAP NAME clause to redirect files as needed.

When restoring files from a BACKUP tape, note the following handling of wild-card characters:

- A file set with an asterisk in the volume, subvolume, and file-identifier positions (*.*.*) directs the RESTORE utility to restore all files on the tape, including local and remote files.
- A file set with a dollar sign in the volume location ($*.*.*) directs the RESTORE utility to restore all of the local files on the tape. Files stored in the remote node format will not be restored or listed using this format.

For example, if you run RESTORE on node \A and specify the file set "$*.*.*" for a tape with a remote file set from node \B, RESTORE does not restore the tape:

```
RESTORE $Tape, ( $*.*.* )

\A
Files not found - Error 2013

$*.*
*      *ERROR -2013* Fileset not dumped (ERROR 11)
```

In the preceding example, you could use the fileset "*.*.*" and an appropriate MAP NAMES option to restore all files to the local node:

```
RESTORE $Tape, ( *.*.* ), MAP NAMES ( *.*.* TO $New.Sub.* )

\A.$New.Sub
 TABLE1          TINDX
```

If you know the remote node name for the file set list, you can specify the node name in the file set (by using the format \<node-name>.$*.*.*) to direct the RESTORE utility to restore all files from that node, as follows;

```
RESTORE $Tape, ( \B.$Vol.*.* ),
MAP NAMES (  \B.$VOL.*.* TO $NEW.SVOL.* )

\A.$New.Sub
 Table1
```

Restoring Files to a Different Network

When restoring files on a node that is not connected to the network where the original BACKUP process was run, RESTORE attempts:

- If all files on the BACKUP tape are in local format, the files are restored to the local node.
- If any files on the BACKUP tape are in remote format, RESTORE attempts to restore the files to the node assigned to the node number stored with the file name.

In this situation, follow these steps before requesting the restore operation:

1. Run the RESTORE utility with the LISTONLY option:

   ```
   RESTORE $Tape, ( *.*.* ), LISTONLY
   ```

   The output describes how the node numbers of the backed-up files match the node names on the destination network. If the source node number exists in the destination network, the output from the LISTONLY option displays the matching node name in the destination network, as follows:

   ```
   $VOL.SUBVOL
     TINDX
   \B.$VOL.SUBVOL
     TABLE1
   ```

2. Run the RESTORE utility again, specifying the appropriate file set in the RESTORE command. Use the MAP NAMES option to specify destination node names. For example, this command restores the files in the previous example:

   ```
   RESTORE $TAPE, ( \B.$VOL.SUBVOL.TABLE1),
   MAP NAMES ( \B.$VOL.SUBVOL.* TO $NEW.SV2.* )
   ```

If the node number does not exist on the destination network, the node name is replaced by “\??” in the LISTONLY output, as follows:

```
$VOL.SUBVOL
  TINDX
\??.$VOL.SUBVOL
  TABLE1
```

△ **Caution.** You cannot currently restore files of a remote file set if their associated node number is not available on the destination network.

For additional examples of the MAP NAMES option in the RESTORE utility, see the *Guardian Disk and Tape Utilities Reference Manual*.

Moving the System Catalog

Although you can move the system catalog to a new location after SQL/MP has been installed, this operation is extremely complex. You should consider moving the system catalog only in special cases.

These instructions are based on the assumptions that the TMF subsystem is operational, that the user is the super ID, and that the SQLCI2 program is on the $SYSTEM.SYSTEM subvolume and is described in the system catalog.

To move the system catalog, follow these steps:

1. Check that no SQL/MP operations or transactions are in progress during the move operation. The system catalog should not be moved on an active system.

2. If your system catalog contains SQL objects, perform a BACKUP of the objects that you want to save, or move the objects to another catalog. You must save the information because the system catalog must be empty for you to drop it. You can use this BACKUP command to save the user-defined objects in the system catalog; enter the command at the command interpreter prompt:

   ```
   20> BACKUP $TAPE, *.*.* FROM CATALOG sys-catalog, AUDITED,
                OPEN, LISTALL
   ```

   In the BACKUP command, *sys-catalog* is the volume and subvolume on which your system catalog resides.

3. If you have only a few objects in the system catalog, purge all objects from the system catalog except for the CATALOGS table and SQLCI2. Remove the objects one at a time by using the DROP statement or PURGE command.

   If the previous method is impractical because you have many objects, save a copy of your SQLCI2 program, and then purge everything in the system catalog (including the SQLCI2 program) with a single command. To save the SQLCI2 program, use DUP to copy the program to a file named ZZSQLCI2. After the PURGE operation, the INITIALIZE SQL command renames and SQL compiles the saved copy of SQLCI2.

   This command sequence accomplishes these operations:

   ```
   21> VOLUME $SYSTEM.SYSTEM
   22> FUP DUP SQLCI2, ZZSQLCI2, SAVEALL
   23> SQLCI
   >> PURGE *.*.* FROM CATALOG sys-catalog, ALLOWERRORS ON;
   >> INITIALIZE SQL;
   ```

   In the PURGE command, *sys-catalog* is the volume and subvolume on which your system catalog resides.

4. Create an OBEY command file you can use to insert the rows from the existing CATALOGS table into the re-created CATALOGS table. This command sequence queries the existing CATALOGS table and generates an OBEY command file that contains a report in INSERT-statement format. This command series appears in OBEY command file format for entering through SQLCI. Use this command file for CATALOGS table versions 300 and later.

```
-- ----------------------------------------------------------------
-- BEGIN OBEY COMMAND FILE COMMAND SEQUENCE TO QUERY THE CATALOGS
-- TABLE AND GENERATE A REPORT IN INSERT-STATEMENT FORMAT
-- ----------------------------------------------------------------
OUT_REPORT obey-insert-file CLEAR;

RESET LAYOUT *;
SET LAYOUT PAGE_LENGTH ALL;

RESET SESSION *;
SET SESSION LIST_COUNT 0;

RESET STYLE *;
SET STYLE HEADINGS OFF;
SELECT CATALOGNAME,
       SUBSYSTEMNAME,
       VERSION,
       VERSIONUPGRADETIME,
       CATALOGCLASS,
       CATALOGVERSION
FROM $catalogs-vol.SQL.CATALOGS
  WHERE CATALOGCLASS <> "S"
;
DETAIL "INSERT INTO $new-catalogs-vol.SQL.CATALOGS", SKIP,
       "  VALUES (""", CATALOGNAME,        """,", SKIP,
       "          """, SUBSYSTEMNAME,      """,", SKIP,
       "          """, VERSION,            """,", SKIP,
       "            ", VERSIONUPGRADETIME, "  ,", SKIP,
       "          """, CATALOGCLASS,       """," , SKIP,
       "          ", CATALOGVERSION,             SKIP,
       "         )", SKIP,
       ";",  SKIP
;
LIST ALL;
-- in which obey-insert-file is any EDIT file;
--  catalogs-vol is the volume on which the current CATALOGS table
--  resides;
--  new-catalogs-vol is the volume on which the new CATALOGS table
is to
--  reside.
OUT_REPORT;                          -- Closes report file
RESET LAYOUT *; RESET STYLE *;       -- Resets report defaults
RESET SESSION *; SET SESSION LIST_COUNT ALL;
-- ----------------------------------------------------------
-- END REPORT PRODUCING COMMAND SERIES -------------------
-- ----------------------------------------------------------
```

For the CATALOGS table versions that are earlier than version 300, use this OBEY command file:

```
-- --------------------------------------------------------------
-- BEGIN OBEY COMMAND FILE COMMAND SEQUENCE TO QUERY THE CATALOGS
-- TABLE AND GENERATE A REPORT IN INSERT-STATEMENT FORMAT
-- --------------------------------------------------------------
OUT_REPORT obey-insert-file CLEAR;

RESET LAYOUT *;
SET LAYOUT PAGE_LENGTH ALL;

RESET SESSION *;
SET SESSION LIST_COUNT 0;

RESET STYLE *;
SET STYLE HEADINGS OFF;
SELECT CATALOGNAME,
       SUBSYSTEMNAME,
       VERSION,
       VERSIONUPGRADETIME,
       CATALOGCLASS
       FROM $catalogs-vol.SQL.CATALOGS
  WHERE CATALOGCLASS <> "S"
;
DETAIL "INSERT INTO $new-catalogs-vol.SQL.CATALOGS", SKIP,
       "  VALUES ("""", CATALOGNAME,        """,", SKIP,
       "          """, SUBSYSTEMNAME,      """,", SKIP,
       "          """, VERSION,            """,", SKIP,
       "            ", VERSIONUPGRADETIME, "  ,", SKIP,
       "          """, CATALOGCLASS,       """" , SKIP,
       "         )", SKIP,
       ";",  SKIP
;
LIST ALL;
-- in which obey-insert-file is any EDIT file;
--  catalogs-vol is the volume on which the current CATALOGS table
--  resides;
--  new-catalogs-vol is the volume on which the new CATALOGS table
is to
--  reside.
OUT_REPORT;                           -- Closes report file
RESET LAYOUT *; RESET STYLE *;        -- Resets report defaults
RESET SESSION *; SET SESSION LIST_COUNT ALL;
-- --------------------------------------------------------
-- END REPORT PRODUCING COMMAND SERIES --------------------
-- --------------------------------------------------------
```

If the CATALOGS table does not contain any entries or you do not have any user catalogs, skip to Step 6 on page 9-12.

5. Set up a licensed SQLCI2L program from a copy of the SQLCI2 program as described in Appendix A, Licensed SQLCI2 Process:

```
33> LOGON SUPER.SUPER, password
34> FUP DUP $SYSTEM.SYSTEM.SQLCI2, $SYSTEM.SYSTEM.SQLCI2L
35> FUP SECURE SQLCI2L, "NN--"
36> SQLCOMP /IN SQLCI2L/ CATALOG $SYSTEM.SQL
37> FUP LICENSE SQLCI2L
```

To enable SQLCI to use the licensed SQLCI2 version rather than the normal SQLCI2 version, you must create the =_SQL_CI2_*sys* DEFINE pointing to the licensed version. This command performs this operation:

```
38> ADD DEFINE =_SQL_CI2_sys, CLASS MAP,
                  FILE $SYSTEM.SYSTEM.SQLCI2L
```

In the ADD DEFINE command, *sys* is the node (system) name without the backslash.

6. After setting up the licensed SQLCI2L process, delete the catalog entries from the CATALOGS table (except for the system catalog entry). The CATALOGS table must be empty before you can drop the existing system catalog; therefore, you must delete any references to user catalogs without affecting the dependent SQL catalogs and objects. Enter this statement using the licensed SQLCI2L process to delete the rows:

   ```
   >> DELETE FROM catalogs-vol.SQL.CATALOGS
   +>     WHERE CATALOGCLASS <> "S";
   ```

   In the DELETE statement, *catalogs-vol* is the volume on which your CATALOGS table resides. The subvolume is always SQL.

7. Save a copy of the SQLCI2 program because the DROP SYSTEM CATALOG command also drops SQLCI2. Enter this FUP DUP command to save a copy of SQLCI2:

   ```
   >> VOLUME $SYSTEM.SYSTEM;
   >> FUP DUP SQLCI2, ZZSQLCI2, SAVEALL;
   ```

8. If you have SQL compiled the licensed SQLCI2L program into your system catalog to perform Step 5 on page 9-11, drop the program so that the system catalog is empty. If you are still using the licensed SQLCI2L program from Step 5 on page 9-11, exit from SQLCI and delete the DEFINE. These commands accomplish this operation:

   ```
   >>  EXIT;
   24>  DELETE DEFINE =_SQL_CI2_sys
   25>  SQLCI
   >>  DROP PROGRAM $SYSTEM.SYSTEM.SQLCI2L;
   ```

   In the DELETE DEFINE command, *sys* is the node (system) name without the backslash.

9. Drop the old system catalog and create the new one in its new location. The system catalog should be empty, but you can query the catalog to verify that all references are dropped, except the tables themselves, including the CATALOGS table.

   To drop the system catalog, you must use the DROP SYSTEM CATALOG command. If you are not running SQLCI, you can drop the catalog from SQLCI by entering these commands:

   ```
   >> DROP SYSTEM CATALOG sys-catalog;
   ```

In the DROP command, *sys-catalog* is the volume and subvolume on which your system catalog resides.

You cannot, however, enter the DROP SYSTEM CATALOG command while SQLCI2 is running, as it normally is when you are running SQLCI and have entered other commands during the current session. If you attempt to enter this command in that case, the command terminates abnormally, and the RDBMS returns an error message. So, you must exit from SQLCI, which implicitly terminates SQLCI2. Next restart SQLCI and enter the DROP SYSTEM CATALOG command at the first SQLCI prompt.

10. Create the new system catalog through SQLCI as follows:

    ```
    >> CREATE SYSTEM CATALOG new-system-catalog;
    ```

    In the command, *new-system-catalog* names the volume and subvolume on which the new system catalog is to reside. The CATALOGS table will be created on the same volume as the system catalog but on the SQL subvolume. For more information on the CREATE SYSTEM CATALOG command, see the *SQL/MP Reference Manual*.

11. Reinitialize SQL/MP through SQLCI as follows:

    ```
    >> INITIALIZE SQL;
    ```

    This operation renames ZZSQLCI2 to SQLCI2, compiles SQLCI2, and registers the SQLCI2 program in the system catalog.

12. Rebuild the CATALOGS table unless the CATALOGS table does not contain any entries or you do not have user catalogs, in which case, skip to Step 14 on page 9-14.

    Add the entries to the CATALOGS table without affecting the dependent SQL catalogs and objects. To accomplish this task, you must be using a licensed SQLCI2 process, as described in Appendix A, Licensed SQLCI2 Process. After setting up the licensed SQLCI2L process, use it to run the OBEY command file (generated by the commands shown in Step 4 on page 9-10) to insert the catalog entries into the CATALOGS table:

    ```
    >> OBEY obey-insert-file;
    ```

    Each INSERT statement in the file (for CATALOGS table versions 300 and later) should look like this sample:

    ```
    INSERT INTO new-catalogs-vol.$SQL.CATALOGS
      VALUES ("\SYS1.$VOL4.INVENT       ",
              "SQL                         ",
              "A345",
               0,
              "U",
               345
             ) ;
    ```

Each INSERT statement in the file (for CATALOGS table earlier than version 300) should look like this sample:

```
INSERT INTO new-catalogs-vol.$SQL.CATALOGS
  VALUES ("\SYS1.$VOL4.INVENT       ",
          "SQL                          ",
          "A011",
           0,
          "U"
          ) ;
```

13. Drop the licensed SQLCI2L process so that you cannot mistakenly use the process. You should also drop the DEFINE for the SQLCI2L file. These commands accomplish these operations:

    ```
    >> EXIT;
    28> DELETE DEFINE =_SQL_CI2_sys
    29> SQLCI
    >> DROP PROGRAM $SYSTEM.SYSTEM.SQLCI2L;
    ```

    In the DELETE DEFINE command, *sys* is the node (system) name without the backslash.

14. If you backed up any SQL objects in Step 2 on page 9-9, restore them to the new system catalog by specifying the CATALOG option pointing to the new system catalog:

    ```
    30> RESTORE $TAPE, *.*.*  CATALOG new-system-catalog,
               AUDITED, OPEN, LISTALL
    ```

    In the RESTORE command, *new-system-catalog* is the volume and subvolume on which your system catalog resides.

Moving Database Objects

You can move individual SQL objects such as tables, views, indexes, and SQL programs stored in Guardian files separately. Usually the dependent objects (shorthand and protection views and indexes) are moved with the underlying table or tables. Comments and constraints are automatically moved with a table.

The DUP and BACKUP/RESTORE utilities support the qualified file-set list to identify the source file list. Qualified file-set list expressions enable you to refine the file set list to specify objects. For information about the qualified file-set list, see DUP and BACKUP/RESTORE on page 9-4. For a more thorough definition of a qualified file-set list, see the *SQL/MP Reference Manual*.

You can move both audited and nonaudited objects either with DUP or with BACKUP and RESTORE. Whenever you move any audited objects, you should include steps to make TMF online dumps of all restored audited objects following the move. The online dumps give the TMF subsystem the current location of the objects for file recovery.

For information about moving an SQL program stored in a Guardian file, see Moving Programs on page 10-39. To move an SQL program stored in an OSS file, use the appropriate OSS utility.

Dropping and Re-creating Catalogs

To move a catalog, you must first drop all the objects described in it, drop the catalog, and then re-create it in a new location. Finally, restore the SQL objects to disk, referring to the new catalog name.

If you want to move both a catalog and all the dependent objects described in that catalog to a new location, see Steps for Moving a Database on page 9-25.

If you want to move only the catalog tables to a new location and keep the dependent objects in the same location, you must use the BACKUP/RESTORE method, because the DUP utility does not allow for the redefinition of a source and target catalog without also mapping target locations for the objects.

Moving Catalogs

To move a catalog, follow these steps:

1. Determine the dependent programs of the objects described in the catalog by using the DISPLAY USE OF command. The move operations invalidate these programs.
2. Back up all the dependent objects described in the catalog, including programs. Use the FROM CATALOGS parameter of the qualified file-set list to identify the objects from the catalog. This command, entered at the command interpreter prompt, accomplishes the operation:

```
43> BACKUP $TAPE, *.*.* FROM CATALOG old-catalog-name,
           AUDITED, OPEN, LISTALL
```

3. Create the new catalog in SQLCI, as shown in this statement:

```
>> CREATE CATALOG new-catalog-name;
```

4. Drop the specified objects and the old catalog:

```
>> PURGE *.*.* FROM CATALOG old-catalog-name ALLOWERRORS ON;
>> DROP CATALOG old-catalog-name;
```

5. Restore the backup tape by using the CATALOG option to define the new catalog. At the command interpreter prompt, enter:

```
45> RESTORE $TAPE, *.*.*, CATALOG new-catalog-name,
            AUDITED, OPEN, TAPEDATE, LISTALL
```

6. SQL compile the programs described in the new catalog or any programs referencing the objects described in the new catalog.
7. Make new TMF online dumps of the catalog and all restored audited objects.

Moving Tables

You can move a table with the SQLCI DUP, LOAD, or COPY utility, or with the Guardian utilities BACKUP and RESTORE. Each utility involves some special considerations.

The DUP and BACKUP/RESTORE utilities enable you to specify the source tables by name or by qualified file-set list. If the table and its dependent objects all reside on the same subvolume and are moved to another subvolume, you can use a target file-set list to specify the new location. If the table and its dependent objects are to reside on two or more subvolumes, you must use the MAP NAMES option. You cannot use the MAP NAMES option and the target file-set list in the same command. If the dependent objects are described in a new catalog, use the CATALOG clause to define the new catalog for the objects.

If you want to move a table to a new subvolume on the same volume or to rename the table, you can use the ALTER TABLE statement with the RENAME option. This approach does not actually transport the data, but alters the directory entry and all associated catalog references to the table.

The move utilities do not automatically move dependent programs with the underlying table. Programs are moved only if they are included in the file set list. After you have moved a table, you should include steps to explicitly SQL compile the dependent programs to avoid automatic recompilation. Programs are not invalidated by the move operation but will be invalidated when the old table is dropped.

The move utilities also do not automatically move any collations used by a table or its dependent objects.

You can move a nonpartitioned table to another volume by using the MOVE option of the ALTER TABLE statement. For more information, see Splitting, Moving, and Merging Partitions on page 7-20. You can also move all or part of a partition of a table. For more information, see Moving Partitions on page 9-23.

Using DUP

When you DUP a table, the utility attempts to duplicate all partitions, indexes, and protection and shorthand views. To duplicate all these objects along with your table, specify either a target file-set list or the MAP NAMES option so that DUP can map the dependent source objects to the target objects. In addition, DUP writes all constraints, comments, and statistical information to the appropriate tables of the target catalog. You can limit the automatic duplication of indexes and views with the INDEXES and VIEWS parameters.

Using COPY and LOAD

You can move tables with the COPY and LOAD utilities. During the LOAD operation, you can define the block structuring of a target key-sequenced table. By specifying SLACK or DSLACK, you can load a source table into a target table with free space for future insertions.

COPY and LOAD do not move dependent objects automatically except any indexes that are defined on the table. Both COPY and LOAD require that you create the new table before copying or loading the data. For additional information on copying or loading tables, see Section 8, Reorganizing Tables and Maintaining Data, or the *SQL/MP Reference Manual*.

Using BACKUP and RESTORE

When you use BACKUP and RESTORE to move a table, the utilities attempt to duplicate all partitions, indexes, and protection views. Shorthand views are duplicated only if you explicitly name them in the file set list. If you use a wild-card string for the file set list, any shorthand view names identified in the wild-card string are treated as if you had explicitly named them.

To move all these objects, specify the MAP NAMES option so that RESTORE can map the dependent source objects to the target objects. In addition, RESTORE writes all constraints, comments, and statistical information to the appropriate tables of the target catalog.

The DUP and BACKUP/RESTORE utilities enable you to specify the source tables by name or by qualified file-set list. If the table and its dependent objects all reside on the same subvolume and are moved to another subvolume, you can use a target file-set list to specify the new location. If the table and its dependent objects are to reside on two or more subvolumes, you must use the MAP NAMES option.

You cannot use the MAP NAMES option and the target file-set list in the same command. If the dependent objects are described in a new catalog, use the CATALOG clause to define the new catalog for the objects.

If you want to move a table to a new subvolume on the same volume or to rename the table, you can use the ALTER TABLE statement with the RENAME option. By using this approach, you do not actually move the data, but you alter the directory entry and all associated catalog references to the table.

The moving utilities do not automatically move dependent programs with the underlying table. Programs are moved automatically only if they are included in the file set list. After you have moved a table, you should include steps to explicitly SQL compile the dependent programs to avoid automatic recompilation. Programs are not invalidated by the moving operation but will be invalidated when the old table is dropped.

Operational Steps

To move a table, follow these steps:

1. Determine the name of the table you want to move.
2. Determine the dependent objects with the DISPLAY USE OF command. Any dependent programs are invalidated when you drop the old table.
3. Check that sufficient space exists on the targeted volumes to create the new table and its dependent objects.
4. Create an OBEY command file for the command if the command is long or will be reused. Consider whether the new table will refer to the same collations as the old table; if not, update the CREATE TABLE statement to refer to the new collations.
5. Check that the table is not in use.
6. Perform the move command of your choice. Use the logging facility of SQLCI or the LISTALL option of BACKUP and RESTORE. Keep the log for your records.
7. Drop the old table by using the DROP TABLE statement.
8. Alter the DEFINEs, if used, to refer to the new location of the table and any dependent views.
9. SQL compile all invalidated programs.
10. Make a new TMF online dump if the table is audited.
11. Restart the application, if stopped, by using the new DEFINEs.

Examples of Using DUP to Move Tables

This example shows how to use the DUP command to move a single table that has no dependent objects from one subvolume to another. The new table is to be registered in the same catalog as the source table. The CATALOG option is required if the current default catalog does not apply. This example sets the default catalog before the DUP command is entered.

```
>>  CATALOG $VOL1.SALES;
>>  DUP $VOL1.SALES.ODETAIL, $VOL1.MKTG.ODETAIL;
```

In this example, the wild-card file name * identifies the target file-set list. By using the wild-card character, you enable a table with dependencies to be moved where the dependent objects are also duplicated. The new table and its dependencies are to be registered in the catalog $VOL1.ADMIN. In this example, all the dependent objects of EMPLOYEE also reside on $VOL1.PERSNL, so a target file-set list is used.

```
>>  DUP $VOL1.PERSNL.EMPLOYEE, $VOL1.ADMIN.*,
+>    CATALOG $VOL1.ADMIN, ALLOWERRORS ON;
```

This example demonstrates the MAP NAMES and CATALOG options. The table EMPLOYEE and its dependencies are being moved from the PERSNL subvolume to the ADMIN subvolume. The MAP NAMES option is used to define the target subvolumes for each object source. The CATALOG option defines the new catalog for each object.

The table and its dependent objects to be moved follow:

EMPLOYEE	A table that resides on $VOL1.PERSNL, described in the catalog $VOL1.PERSNL
EMPLIST	A protection view that resides on the same subvolume, described in the same catalog as its underlying table, EMPLOYEE
XEMPNAME	An index that resides on $VOL2.PERSNL, described in the catalog $VOL2.PERSNL

This DUP command moves these objects:

```
>> DUP $VOL1.PERSNL.EMPLOYEE,
+>    MAP NAMES ($VOL1.PERSNL.* TO $VOL1.ADMIN.*,
+>               $VOL2.PERSNL.* TO $VOL2.ADMIN.*)
+>    CATALOG ($VOL1.ADMIN FOR $VOL1.ADMIN.EMPLOYEE,
+>             $VOL1.ADMIN FOR $VOL1.ADMIN.EMPLIST,
+>             $VOL2.ADMIN FOR $VOL2.ADMIN.XEMPNAME),
+>    ALLOWERRORS ON;
```

This example demonstrates the use of the INDEXES OFF and VIEWS OFF options. You can use these options to duplicate a table with dependent views and indexes without automatically duplicating those dependent views and indexes.

```
>>  DUP $VOL1.PERSNL.EMPLOYEE, $VOL1.ADMIN.*,
+>    CATALOG $VOL1.ADMIN, INDEXES OFF, VIEWS OFF;
```

Examples of Using BACKUP and RESTORE to Move Tables

This example shows how to use BACKUP and RESTORE from an OBEY command file on a single nonaudited table with no dependent objects. BACKUP writes the table to tape. RESTORE moves the table from one subvolume to another and describes it in a new catalog.

```
BACKUP $TAPE, $VOL1.PERSNL.MANAGERS, LISTALL

RESTORE $TAPE, $VOL1.PERSNL.MANAGERS,&
        MAP NAMES ($VOL1.PERSNL.MANAGERS TO $VOL1.ADMIN.MANAGERS),&
        CATALOG ($VOL1.ADMIN FOR $VOL1.ADMIN.MANAGERS),&
        TAPEDATE, LISTALL
```

This example shows how to use BACKUP and RESTORE from an OBEY command file on an audited table with dependent objects that reside on different volumes. You must use the MAP NAMES option to map the dependent objects correctly to new subvolumes. You must also use the CATALOG option to specify the new catalog for each object.

The table and dependent objects to be moved follow:

EMPLOYEE	A table that resides on $VOL1.PERSNL, described in the catalog $VOL1.PERSNL
EMPLIST	A protection view that resides on the same subvolume, described in the same catalog as its underlying table
XEMPNAME	An index that resides on $VOL2.PERSNL, described in the catalog $VOL2.PERSNL

These commands accomplish the move operation:

```
BACKUP $TAPE, $VOL1.PERSNL.EMPLOYEE, AUDITED,LISTALL

RESTORE $TAPE, *.*.*,
   MAP NAMES ($VOL1.PERSNL.* TO $VOL1.ADMIN.*,&
              $VOL2.PERSNL.* TO $VOL2.ADMIN.*),&
   CATALOG ($VOL1.ADMIN FOR $VOL1.ADMIN.EMPLOYEE,&
            $VOL1.ADMIN FOR $VOL1.ADMIN.EMPLIST,&
            $VOL2.ADMIN FOR $VOL2.ADMIN.XEMPNAME),&
   AUDITED, TAPEDATE, LISTALL
```

Examples of Using LOAD or COPY to Move a Table

The next two examples compare using LOAD and COPY in moving SQL tables and dependent objects.

The table and its dependent objects follow:

EMPLOYEE	A table that resides in $OLD.PERSNL, described in the catalog $OLD.PERSNL
EMPLIST	A protection view that resides in the same subvolume, described in the same catalog as its underlying table, EMPLOYEE
XEMPNAME	An index that resides in $OLD2.PERSNL, described in the catalog $OLD2.PERSNL

The first example shows how to create the appropriate dependent structure so that dependent objects are loaded correctly with the table. The CREATE TABLE and CREATE INDEX operations precede the load. Auditing of the table is disabled before the LOAD operation and enabled after the LOAD operation. The SLACK option of the LOAD command specifies the amount of empty space in the block structure. Because a view is only a definition and does not require loading, you could create the view EMPLIST either before or after the LOAD operation.

```
>> CREATE TABLE $VOL1.PERSNL.EMPLOYEE LIKE $OLD.PERSNL.EMPLOYEE
+>     WITH CONSTRAINTS;
>> CREATE INDEX $VOL2.PERSNL.XEMPNAME
+>     ON $VOL1.PERSNL.EMPLOYEE (LAST_NAME, FIRST_NAME)
+>     CATALOG $VOL2.PERSNL;
>> ALTER TABLE $VOL1.PERSNL.EMPLOYEE NO AUDIT;
>> LOAD $OLD.PERSNL.EMPLOYEE, $VOL1.PERSNL.EMPLOYEE,
+>        SORTED, SLACK 20;
>> ALTER TABLE $VOL1.PERSNL.EMPLOYEE AUDIT;
>> CREATE VIEW $VOL1.PERSNL.EMPLIST
+>     AS SELECT
+>        EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, JOBCODE
+>     FROM $VOL1.PERSNL.EMPLOYEE
+>     FOR PROTECTION
+>     CATALOG $VOL1.PERSNL;
```

The LOAD utility automatically loads any dependent indexes created before loading the underlying table. The CREATE TABLE LIKE statement, however, does not apply associated partitions, views, or indexes to the target table. You must create these after creating the table.

The next example shows a sequence of statements and commands to copy the EMPLOYEE table and its dependent index and view. If the target table is empty, this COPY example and the preceding LOAD example create the same files except for the slack space created with the load operation. The COPY operation occurs within a user-defined TMF transaction.

```
>> CREATE TABLE $VOL1.PERSNL.EMPLOYEE LIKE $OLD.PERSNL.EMPLOYEE
+>     WITH CONSTRAINTS;
>> CREATE INDEX $VOL2.PERSNL.XEMPNAME
+>     ON $VOL1.PERSNL.EMPLOYEE (LAST_NAME, FIRST_NAME)
+>     CATALOG $VOL2.PERSNL;
>> BEGIN WORK;
>> COPY $OLD.PERSNL.EMPLOYEE, $VOL1.PERSNL.EMPLOYEE;
>> COMMIT WORK;
>> CREATE VIEW $VOL1.PERSNL.EMPLIST
+>     AS SELECT
+>        EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, JOBCODE
+>     FROM $VOL1.PERSNL.EMPLOYEE
+>     FOR PROTECTION
+>     CATALOG $VOL1.PERSNL;
```

Moving Views

Views do not maintain any physical data to be moved, but they do maintain physical file labels. Typically, protection views are implicitly moved when the underlying table is moved. Dependencies between a protection view and the underlying table are enforced by SQL/MP. Shorthand views can be moved with normal BACKUP and RESTORE procedures.

The DUP utility enables you to implicitly move both protection and shorthand views when you move the underlying table. You can restrict whether a view is duplicated by using the VIEWS EXPLICIT option. If you explicitly name a view in the file set list and specify the VIEWS EXPLICIT option, the view is moved. If the file set list does not include the view but you do specify VIEWS EXPLICIT, the view is not moved automatically. You can specify not to automatically duplicate views with the underlying table by including the VIEWS OFF option.

The BACKUP and RESTORE utilities always move protection views when moving the underlying table. Shorthand views are not automatically moved unless they are explicitly named in the file set list. No options apply to BACKUP and RESTORE to restrict the automatic move of protection views with the underlying table.

In cases where you need to move a large number of views, it will be easier for you to create the views on the target instead of using the BACKUP and RESTORE utilities to move them.

Invalid Views

The possibility that a shorthand view definition might refer to tables or views that have not yet been moved to the new location means that the view definition might be created but marked invalid. After all the objects have been moved, the utilities attempt to make the view valid. If the utility does not complete for any reason or if you specified an invalid mapping scheme, a view might be left in an invalid state. If a view is invalid after a move operation, you must drop the view and re-create it. For more information about invalidity, see Program Validity on page 10-1.

If you want to move a view to a new subvolume on the same volume or rename a view, use the ALTER VIEW statement with the RENAME option. This approach alters the directory entry and all associated catalog references to the view.

Shorthand views and protection views contain no physical data; therefore, there is an easy way to move the view: re-create the view definition at the new location, registering the view in a catalog, then drop the view definition at the old location.

Example

This example moves a view by re-creating the view in a new location and then dropping the old view:

```
>> CREATE VIEW new-view ...;
>> DROP VIEW old-view;
```

Moving Indexes

An index moves with the underlying table. The DUP utility and the BACKUP/RESTORE utilities enable you to move the underlying table and indexes to a new location by using the MAP NAMES option. You can move an index on the same volume with the ALTER INDEX RENAME statement. Furthermore, you can move an index to another volume by re-creating the index on the target volume with the CREATE INDEX statement and then dropping the index from the source volume with the DROP statement.

The DUP utility enables you to either implicitly move all indexes with the underlying table or to turn the implicit index duplication off. If you specify INDEXES OFF, no indexes are moved with the underlying table. DUP implicitly moves indexes with the underlying table by default or when you specify the INDEXES IMPLICIT option, regardless of whether the indexes are named in the file set list.

The BACKUP and RESTORE utilities enable you to either implicitly move all indexes with the underlying table or to move only the indexes explicitly named in the source file-set list. If you explicitly name the index in the file set list and include the INDEXES EXPLICIT option, the index is moved. If you do not include the index in the file set list but include the INDEXES EXPLICIT option, the index is not moved. BACKUP and RESTORE implicitly move indexes with the underlying table by default or when you specify the INDEXES IMPLICIT option, regardless of whether they are named in the file set list.

If you want to move an index to a new subvolume on the same volume or to rename the index, you can use the ALTER INDEX statement with the RENAME option. This strategy does not actually move the data, but alters the directory entry and all associated catalog references to the index.

If you want to move an index from one location to another without moving the underlying table, re-create the index in the new location, registering it in a catalog, and then drop the old index.

This example moves an index by re-creating the index in a new location and then dropping the old index:

```
>> CREATE INDEX new-index ...;
>> DROP INDEX old-index;
```

You can also move all or part of a partition of an index. For more information, see Moving Partitions on page 9-23.

Moving Collations

If you move a collation, any dependent objects that refer to the original collation do not refer to the moved collation. You can move a collation by copying it with the SQLCI DUP utility or the Guardian BACKUP and RESTORE utilities. If you want dependent objects to refer to the new collation, you must drop and re-create the dependent objects, with references to the moved collation.

Moving Partitions

To move, split, merge, or redefine row boundaries of partitions of tables and indexes, use the PARTONLY MOVE option of the ALTER TABLE or ALTER INDEX statement. To minimize interruptions to data availability during the operation, use the WITH SHARED ACCESS option.

For more information, see Splitting, Moving, and Merging Partitions on page 7-20 and the descriptions of the ALTER TABLE and ALTER INDEX statements in the *SQL/MP Reference Manual*.

Moving a Database to a Different Node or Different Volumes

Moving a database involves moving a set of SQL objects from one environment to another. This move might be of database objects from one group to another group or of an entire database from one node (system) to another node.

This discussion examines a scenario where you want to move a complete grouping of SQL objects defined in one or more catalogs. Suppose that you also want the database to retain the same consistent state in the new location as it has in the old location.

Choosing a Method

You might choose the DUP method or the BACKUP/RESTORE method of moving objects. Your choice depends on the location of the two environments and the available media for transfer.

You might choose the BACKUP/RESTORE method in these cases:

- The target volumes do not have enough space to maintain two copies of the objects at the same time. DUP requires both objects to be online simultaneously until the operation is complete.
- The source and target locations are not physically connected.
- You want to maintain a backup copy of the database on tape media.
- The database objects might need to be restored multiple times; for example, the database environments are released or moved to a group to do testing or documentation. The tape method gives you an archive copy so you can restore the database many times to the same consistent state.

If you have SQL programs stored in OSS files in your database, use OSS utilities to move these programs separately and then recompile them on the new node. For more information about OSS utilities, see the *Open System Services Shell and Utilities Reference Manual*.

Moving databases can be very complicated if the database files and catalog layout is intricate. You must also have the proper read authority to the source objects and catalogs, and you must have write authority to the target catalogs.

Steps for Moving a Database

To move a database, follow these steps, which are demonstrated in Example 9-1 on page 9-26:

1. Determine the name of the SQL catalogs involved in this move by querying the CATALOGS table.
2. Determine the names of the SQL objects involved in this move by querying the catalogs determined in Step 1.
3. Determine the dependencies with the DISPLAY USE OF command or with catalog queries. It is important that you consider the interdependencies of the database for the move.
4. If you need a consistent copy of the database, you should make sure the SQL objects are not in use.
5. Perform the DUP or BACKUP/RESTORE command.
6. Verify the status of the database with the VERIFY utility or with catalog queries.
7. SQL compile the programs (by using SQLCOMP for SQL programs that run in the Guardian environment or c89 for SQL programs that run in the OSS environment).
8. Make new TMF online dumps of all restored catalogs and audited objects.

Example of Moving a Database

The sample database released with the software is the database used in the example of moving a database shown in Example 9-1 on page 9-26. This example shows moving the database from volume $DATA to $DATA1 by using the DUP utility. The example also shows the necessary steps and information required for moving a database. In the example, descriptions of the steps for moving the database appear in boldface text.

Example 9-1. Example of Moving the Sample Database (page 1 of 5)

1. Determine the name of the SQL catalogs involved in this move by querying the CATALOGS table. You should always log the commands and information returned in the move operations.

```
>> LOG log-file CLEAR;
>> SELECT CATALOGNAME FROM $SYSTEM.SQL CATALOGS;

CATALOGNAME
------------------------
\SYS1.$DATA.INVENT
\SYS1.$DATA.PERSNL
\SYS1.$DATA.SALES
```

2. Determine the names of the SQL objects involved in this move. The TABLES catalog table can be queried to obtain a list of tables. In the case of the sample database, the TABLECODE for the application tables is 0. This table code might not always be applicable.

```
>> SELECT TABLENAME FROM INVENT.TABLES
+>     WHERE TABLECODE = 0;
TABLENAME
----------------------------------

\SYS1.$DATA.INVENT.ERRORS
\SYS1.$DATA.INVENT.PARTLOC
\SYS1.$DATA.INVENT.PARTSUPP
\SYS1.$DATA.INVENT.SUPPLIER
\SYS1.$DATA.INVENT.VIEW207

--- 5 row(s) selected.

>> SELECT TABLENAME FROM PERSNL.TABLES
+>     WHERE TABLECODE = 0;

TABLENAME
----------------------------------
\SYS1.$DATA.PERSNL.DEPT
\SYS1.$DATA.PERSNL.EMPLIST
\SYS1.$DATA.PERSNL.EMPLOYEE
\SYS1.$DATA.PERSNL.JOB
\SYS1.$DATA.PERSNL.MGRLIST

--- 5 row(s) selected.
>> SELECT TABLENAME FROM SALES.TABLES
+>     WHERE TABLECODE = 0;
TABLENAME
----------------------------------
\SYS1.$DATA.SALES.CUSTLIST
\SYS1.$DATA.SALES.CUSTOMER
\SYS1.$DATA.SALES.ODETAIL
\SYS1.$DATA.SALES.ORDERS
\SYS1.$DATA.SALES.ORDREP
\SYS1.$DATA.SALES.PARTS

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

Example 9-1. Example of Moving the Sample Database (page 2 of 5)

```
3.  Determine the dependencies with the DISPLAY USE OF command or with
    catalog queries.
    The layout of the dependencies determines the structure of the correct
    DUP command used in Step 5. For more information about interpreting
    dependencies, see Catalog Mapping Schemes for DUP on page 9-30.

4.  If you need a consistent copy of these tables, check that the SQL objects
    are not in use.

5.  Perform the DUP or BACKUP/RESTORE command.
    This example uses the DUP command, but you can use either utility. The
    DUP command requires that you create new catalogs before you perform the
    move operation. You might want to create an OBEY command file for the
    following statements and commands.
```

```
>>CREATE CATALOG $DATA1.INVENT;
--- SQL operation complete.
>>CREATE CATALOG $DATA1.PERSNL;
--- SQL operation complete.
>>CREATE CATALOG $DATA1.SALES;
--- SQL operation complete.
>>DUP ($DATA.INVENT.*, $DATA.PERSNL.*, $DATA.SALES.*),
+>  MAP NAMES ( $DATA.INVENT.* TO $DATA1.INVENT.*,
+>              $DATA.PERSNL.* TO $DATA1.PERSNL.*,
+>              $DATA.SALES.* TO $DATA1.SALES.*),

+>  CATALOG ($DATA1.INVENT FOR $DATA1.INVENT.*,
+>           $DATA1.PERSNL FOR $DATA1.PERSNL.*,
+>           $DATA1.SALES FOR $DATA1.SALES.*),
+>ALLOWERRORS ON, LISTALL;
DUPLICATED TABLE  $DATA.INVENT.ERRORS TO $DATA1.INVENT.ERRORS
DUPLICATED TABLE  $DATA.INVENT.PARTLOC TO
                  $DATA1.INVENT.PARTLOC
DUPLICATED TABLE  $DATA.INVENT.PARTSUPP TO
                  $DATA1.INVENT.PARTSUPP
           INDEX  $DATA.INVENT.XSUPORD TO
                  $DATA1.INVENT.XSUPORD
DUPLICATED TABLE  $DATA.INVENT.SUPPLIER TO
                  $DATA1.INVENT.SUPPLIER
           INDEX  $DATA.INVENT.XSUPPNAM TO
                  $DATA1.INVENT.XSUPPNAM
DUPLICATED TABLE  $DATA.PERSNL.DEPT TO $DATA1.PERSNL.DEPT
           INDEX  $DATA.PERSNL.XDEPTMGR TO
                  $DATA1.PERSNL.XDEPTMGR
           INDEX  $DATA.PERSNL.XDEPTRPT TO
                  $DATA1.PERSNL.XDEPTRPT
DUPLICATED TABLE  $DATA.PERSNL.EMPLOYEE TO
                  $DATA1.PERSNL.EMPLOYEE
```

Example 9-1. Example of Moving the Sample Database (page 3 of 5)

```
   PVIEW  $DATA.PERSNL.EMPLIST TO
                 $DATA1.PERSNL.EMPLIST
          INDEX  $DATA.PERSNL.XEMPNAME TO
                 $DATA1.PERSNL.XEMPNAME
          INDEX  $DATA.PERSNL.XEMPDEPT TO
                 $DATA1.PERSNL.XEMPDEPT
DUPLICATED TABLE  $DATA.PERSNL.JOB TO $DATA1.PERSNL.JOB
DUPLICATED TABLE  $DATA.SALES.CUSTOMER TO
                 $DATA1.SALES.CUSTOMER
          PVIEW  $DATA.SALES.CUSTLIST TO
                 $DATA1.SALES.CUSTLIST
          INDEX  $DATA.SALES.XCUSTNAM TO
                 $DATA1.SALES.XCUSTNAM
DUPLICATED TABLE  $DATA.SALES.ODETAIL TO $DATA1.SALES.ODETAIL
DUPLICATED TABLE  $DATA.SALES.ORDERS TO $DATA1.SALES.ORDERS
          INDEX  $DATA.SALES.XORDREP TO $DATA1.SALES.XORDREP
          INDEX  $DATA.SALES.XORDCUS TO $DATA1.SALES.XORDCUS
DUPLICATED TABLE  $DATA.SALES.PARTS TO $DATA1.SALES.PARTS
          INDEX  $DATA.SALES.XPARTDES TO
                 $DATA1.SALES.XPARTDES
          SVIEW  $DATA.INVENT.VIEW207 TO
                 $DATA1.INVENT.VIEW207
          SVIEW  $DATA.PERSNL.MGRLIST TO
                 $DATA1.PERSNL.MGRLIST
          SVIEW  $DATA.SALES.ORDREP TO $DATA1.SALES.ORDREP
26 OBJECT(S) DUPLICATED
   catalog queries.

   >> VERIFY $DATA1.INVENT.*;
      ...                        <-- omitted miscellaneous tables
   --- Verifying $DATA1.INVENT.ERRORS
   --- $DATA1.INVENT.ERRORS verified.
   --- Verifying $DATA1.INVENT.PARTLOC
   --- $DATA1.INVENT.PARTLOC verified.
   --- Verifying $DATA1.INVENT.PARTSUPP
   --- $DATA1.INVENT.PARTSUPP verified.
   --- Verifying $DATA1.INVENT.SUPPLIER
   --- $DATA1.INVENT.SUPPLIER verified.
   --- Verifying $DATA1.INVENT.VIEW207
   --- $DATA1.INVENT.VIEW207 verified.
   --- Verifying $DATA1.INVENT.XSUPORD
   --- $DATA1.INVENT.XSUPORD verified.
   --- Verifying $DATA1.INVENT.XSUPPNAM
   --- $DATA1.INVENT.XSUPPNAM verified.
   --- SQL operation complete.
```

Example 9-1. Example of Moving the Sample Database (page 4 of 5)

```
>> VERIFY $DATA1.PERSNL.*;
   ...                     <-- omitted miscellaneous tables
--- Verifying $DATA1.PERSNL.DEPT
--- $DATA1.PERSNL.DEPT verified.
--- Verifying $DATA1.PERSNL.EMPLIST
--- $DATA1.PERSNL.EMPLIST verified.
--- Verifying $DATA1.PERSNL.EMPLOYEE
--- $DATA1.PERSNL.EMPLOYEE verified.
--- Verifying $DATA1.PERSNL.JOB
--- $DATA1.PERSNL.JOB verified.
--- Verifying $DATA1.PERSNL.MGRLIST
--- $DATA1.PERSNL.MGRLIST verified.
--- Verifying $DATA1.PERSNL.XDEPTMGR
--- $DATA1.PERSNL.XDEPTMGR verified.
--- Verifying $DATA1.PERSNL.XDEPTRPT
--- $DATA1.PERSNL.XDEPTRPT verified.
--- Verifying $DATA1.PERSNL.XEMPDEPT
--- $DATA1.PERSNL.XEMPDEPT verified.
--- Verifying $DATA1.PERSNL.XEMPNAME
--- $DATA1.PERSNL.XEMPNAME verified.
--- SQL operation complete.
>> VERIFY $DATA1.SALES.*;
   ...                     <-- omitted miscellaneous tables
--- Verifying $DATA1.SALES.CUSTLIST
--- $DATA1.SALES.CUSTLIST verified.
--- Verifying $DATA1.SALES.CUSTOMER
--- $DATA1.SALES.CUSTOMER verified.
--- Verifying $DATA1.SALES.ODETAIL
--- $DATA1.SALES.ODETAIL verified.
--- Verifying $DATA1.SALES.ORDERS
--- $DATA1.SALES.ORDERS verified.
--- Verifying $DATA1.SALES.ORDREP
--- $DATA1.SALES.ORDREP verified.
--- Verifying $DATA1.SALES.PARTS
--- $DATA1.SALES.PARTS verified.
--- Verifying $DATA1.SALES.XCUSTNAM
--- $DATA1.SALES.XCUSTNAM verified.
--- Verifying $DATA1.SALES.XORDCUS
--- $DATA1.SALES.XORDCUS verified.
--- Verifying $DATA1.SALES.XORDREP
--- $DATA1.SALES.XORDREP verified.
--- Verifying $DATA1.SALES.XPARTDES
--- $DATA1.SALES.XPARTDES verified.
--- SQL operation complete.
>> EXIT
End of SQLCI Session
```

Example 9-1. Example of Moving the Sample Database (page 5 of 5)

```
7.  SQL compile the programs.
    To validate the programs, you should explicitly SQL compile them with the
    DEFINEs pointing to the new location of the tables and views. The
    following commands demonstrate setting the new DEFINEs and compiling a
    set of SQL programs that run in the Guardian environment. The sample
    database programs reside in the SAMPDB subvolume.

   SET DEFINE CLASS CATALOG
   ADD DEFINE =INVENT , SUBVOL $DATA1.INVENT
   SET DEFINE CLASS MAP
   ADD DEFINE =PARTS   , FILE $DATA1.SALES.PARTS
   ADD DEFINE =SUPPLIER, FILE $DATA1.INVENT.SUPPLIER
   ADD DEFINE =PARTSUPP, FILE $DATA1.INVENT.PARTSUPP
   VOLUME $DATA1.SAMPDB
   SQLCOMP  /IN OBJ205, OUT $S.#HOLD / CATALOG =INVENT,
       EXPLAIN DEFINES
   SQLCOMP  /IN OBJ206, OUT $S.#HOLD / CATALOG =INVENT,
       EXPLAIN DEFINES

8. Make new TMF online dumps of all restored audited objects.
```

Catalog Mapping Schemes for DUP

The structure of the MAP NAMES and CATALOG clauses of the DUP command depend on the structure of database dependencies. You can use the DISPLAY USE OF command or queries of catalog tables to determine the dependencies. These guidelines apply:

- The current location of the objects determines the source file-set list for the DUP command.
- The source and target locations of objects to be moved determines the MAP NAMES layout.
- For the DUP command to automatically duplicate all the dependent objects, the source file-set list must include all tables to be duplicated or all subvolumes that contain these tables.

The DUP command must specify a valid mapping scheme for each object listed in the file set list and all the dependencies of the objects listed; these objects will be automatically duplicated. You must analyze the output of the DISPLAY USE OF utility to determine whether the mapping scheme will identify a valid source to target mapping scheme for all objects. An invalid mapping scheme causes errors, invalid dependent objects, or objects that are not moved.

These examples explain the relationship between the database layout as produced by the DISPLAY USE OF utility and the layout specified in the DUP command.

This DUP command uses a wild-card character for files in the source file-set list to specify that all objects residing on $VOL1.PERSNL are to be duplicated to a new volume. All the dependent objects also reside on $VOL1.PERSNL, so the MAP NAMES specification can use the wild-card character to identify the target location for all objects. All of the objects in this example are described in the catalog $VOL2.PERSNL.

```
>> DUP $VOL1.PERSNL.*,
+>   MAP NAMES ($VOL1.PERSNL.* TO $VOL2.PERSNL.*),
+>   CATALOG ($VOL2.PERSNL FOR $VOL2.PERSNL.*);
```

This example is similar to the preceding example except that the objects identified by the source file-set list $VOL1.PERSNL.* have dependent objects that reside on other volumes. A mapping scheme must be specified that includes both the objects on $VOL1.PERSNL and the dependent objects on $VOL3.PERSNL and $VOL5.PERSNL.

```
>> DUP $VOL1.PERSNL.*,
+>   MAP NAMES ($VOL1.PERSNL.* TO $VOL2.PERSNL.*,
+>              $VOL3.PERSNL.* TO $VOL4.PERSNL.*,
+>              $VOL5.PERSNL.* TO $VOL6.PERSNL.*),
+>   CATALOG ($VOL2.PERSNL);
```

This example shows a DUP command in which the source file-set list specifies a qualified file-set list with the FROM CATALOG specification. All objects described in the catalog $VOL1.PERSNL are to be duplicated to a new volume and described in a new catalog on $VOL2. The source objects could reside anywhere on the node. This DUP command must include a detailed MAP NAMES specification that identifies each source object and its target location.

```
>> DUP (*.*.* FROM CATALOG $VOL1.PERSNL),
+>   MAP NAMES ($VOL1.PERSNL.EMPLOYEE TO $VOL2.PERSNL.EMPLOYEE,
+>              $VOL1.PERSNL.XEMPNAME TO $VOL2.PERSNL.XEMPNAME,
+>              $VOL1.SALES.XEMPSLM TO $VOL2.SALES.XEMPSLM,
+>              $VOL2.SALES.ORDREP TO $VOL4.SALES.ORDREP),
+>   CATALOG ($VOL2.PERSNL FOR $VOL2.PERSNL.EMPLOYEE,
+>            $VOL2.PERSNL FOR $VOL2.PERSNL.XEMPNAME,
+>            $VOL2.PERSNL FOR $VOL2.SALES.XEMPSLM,
+>            $VOL2.PERSNL FOR $VOL4.SALES.ORDREP);
```

The DUP command must specify a catalog for each object. If all objects in a subvolume are described in one catalog, the command can use a simple file-set list for each CATALOG specification. If objects residing in the same subvolume are described in different catalogs, the command must use a CATALOG specification to identify a catalog for each object. The target catalogs must exist when the DUP command executes.

This example shows a DUP command in which the CATALOG specification is mapped to a simple file-set list. All objects residing on $VOL2.PERSNL are to be described in the same catalog.

```
>> DUP $VOL1.PERSNL.*,
+>   MAP NAMES ($VOL1.PERSNL.* TO $VOL2.PERSNL.*),
+>   CATALOG ($VOL2.PERSNL FOR $VOL2.PERSNL.*);
```

This example shows a DUP command in which the CATALOG specification identifies individual objects. The objects residing on $VOL2.PERSNL are to be described in several catalogs.

```
>> DUP $VOL1.PERSNL.*,
+>   MAP NAMES ($VOL1.PERSNL.* TO $VOL2.PERSNL.*),
+>   CATALOG ($VOL2.PERSNL FOR $VOL2.PERSNL.EMPLOYEE,
+>            $VOL2.PERSNL FOR $VOL2.PERSNL.XEMPNAME,
+>            $VOL2.SALES FOR $VOL2.PERSNL.ORDREP);
```

Renaming or Renumbering a Node

Choose your node name and number carefully to avoid the need to change them in the future. The node name is expanded in the catalog entries, and the node number is entered in the file labels throughout the database. SQL/MP relies on the file names and node numbers of SQL objects. If you rename or renumber a node without changing the file labels, SQL generates an error the next time one of the SQL objects on that node is used or referenced.

A change to a node name or number in a distributed environment requires one of these procedures:

- Backing up, purging, and restoring these SQL objects:
    - All SQL noncatalog objects on the affected node
    - All SQL objects that have a partition on the affected node
    - All SQL objects that refer to an object on the affected node, such as dependent objects (views, indexes, constraints)

Note. To verify that the file labels of SQL objects contain the correct internal file names, you must back up and purge *all* SQL objects from a node before the node is renumbered and then restore the objects after the renumbering operation. Alternately, use the MODIFY DICTIONARY utility, described next.

- Using the MODIFY DICTIONARY utility to change the node name or node number or to register a user catalog in the local system catalog. Before using the MODIFY DICTIONARY utility, see the *Nomadic Disk Subsystem User’s Guide* for a detailed description of the process, including coordination with TMF operations. The general steps are:
    - Backing up these SQL objects:
        - All SQL non-catalog objects on the affected node
        - All SQL objects that have a partition on the affected node
        - All SQL objects that refer to an object on the affected node, such as dependent objects (views, indexes, constraints)

    - Using the MODIFY DICTIONARY utility to make the necessary changes, as follows:
        - Use the MODIFY DICTIONARY CATALOG command to change node names in catalogs on the local node.
        - Use the MODIFY DICTIONARY LABEL command to change node numbers in file labels of SQL objects and SQL object programs on the local node.
        - Use the MODIFY DICTIONARY REGISTER command to register a user-defined catalog in the local system catalog.
- Moving all SQL objects and all partitions of SQL objects from the affected node to another node

The time needed to restore an SQL/MP database depends on the size of the database and could be substantial. You can reduce the time by removing all unwanted SQL objects before you back up the database.

Backing Up and Purging SQL Objects

Before renaming or renumbering a node, follow these steps to back up and purge SQL objects. Each step is explained later in detail.

1. Find all the objects on the local node and on remote nodes that need to be backed up.
2. Back up the SQL programs by using the appropriate utility depending on whether the program is stored in a Guardian or OSS file.
3. Back up all other SQL objects except catalogs by using the BACKUP utility.
4. Create an OBEY command file that will re-create the catalogs with the same ownership and security as the original catalogs and catalog tables.
5. Purge the SQL objects and programs.
6. Drop all user catalogs.
7. Back up and drop the system catalog, removing SQL/MP from the node.
8. Check for and purge any detached SQL objects (objects not registered in a catalog).

Finding Objects to Be Backed Up (Step 1)

The SQL/MP system catalog contains the names of all catalogs on the node, in the CATALOGNAME column of the CATALOGS table. Use the SQLCI FILEINFO command to find the volume name of the system catalog:

```
>> FILEINFO $*.SQL.CATALOGS;
```

Then, use the SELECT statement to list the catalog names:

```
>> LOG log-file CLEAR;
>> SELECT CATALOGNAME FROM $volume.SQL.CATALOGS;
```

In the example, `log-file` is a device, process, or disk file, and `$volume` is the volume on which the system catalog resides.

To find all local and remote objects that need to be backed up and the catalog names of dependent objects, query the USAGES table in each catalog:

```
>> SELECT * FROM catalog-name.USAGES;
```

All user-defined objects listed in the USAGES tables of the catalogs on the affected node need to be backed up and purged before the renaming or renumbering operation. For example, if you have an index on a node you want to renumber, you must back up the underlying table even if the table does not have a single partition on the node.

Do not attempt to back up and restore catalog tables and indexes; those objects are re-created when the catalogs are re-created.

Backing Up SQL Programs (Step 2)

To back up SQL programs stored in Guardian files on your node, use the Guardian BACKUP utility. Back up all the programs registered in each catalog by using the FROM CATALOG option to back up all programs in a catalog onto tape.

To back up an SQL program stored in an OSS file, use the appropriate OSS utility.

Backing up SQL programs separately from SQL objects is recommended. You can issue BACKUP commands at a TACL prompt, but it is probably more efficient to do the BACKUP operation by using an OBEY command file.

This command backs up all SQL programs registered in the specified catalog:

```
53> BACKUP $TAPE1, *.*.*
    FROM CATALOG old-catalog-name WHERE SQLPROGRAM,
    ARCHIVEFORMAT, AUDITED, OPEN, LISTALL
```

Use the AUDITED option for all objects, including nonaudited ones, because the file labels for the objects are audited.

Backing Up Other SQL Objects (Step 3)

This command backs up all SQL objects other than programs registered in the specified catalog:

```
54> BACKUP $TAPE2, *.*.*
    FROM CATALOG old-catalog-name WHERE NOT SQLPROGRAM,
    ARCHIVEFORMAT, AUDITED, OPEN, LISTALL
```

Note. Do not back up user catalogs, because RESTORE cannot restore them as user catalogs. Use an OBEY command file instead to re-create the catalogs and the catalog security and ownership.

Creating an OBEY Command File to Re-Create the Catalogs (Step 4)

Create an OBEY command file containing SQL statements that will re-create your catalogs. The statements must specify the same security and the same owners for each catalog and for each catalog table that can be individually secured.

The catalog security is the security of the catalog tables, except for the USAGES, TRANSIDS, and PROGRAMS tables, which can be secured individually. In the system catalog, the CATALOGS table can also be secured individually.

To find out the security and owner of your catalog tables, query the catalog TABLES table:

```
>> LOG log-file CLEAR;
>> SELECT TABLENAME, SECURITYVECTOR, GROUPID, USERID
+>   FROM catalog-name.TABLES
+>   WHERE TABLENAME = "\system.$volume.catalog-name.TABLES";
```

To find out the security and owner of the USAGES, TRANSIDS, and PROGRAMS tables, specify those tables in a query:

```
>> SELECT TABLENAME, SECURITYVECTOR, GROUPID, USERID
+>   FROM catalog-name.TABLES
+>   WHERE TABLENAME = "\system.$volume.catalog-name.USAGES"
+>   OR TABLENAME = "\system.$volume.catalog-name.TRANSIDS"
+>   OR TABLENAME = "\system.$volume.catalog-name.PROGRAMS";
```

The OBEY command file needs to contain these statements:

- A CREATE CATALOG statement for each catalog on the node. To make sure the catalog is re-created with the same security, use the SECURE option to specify the catalog security:

  ```
  >> CREATE CATALOG $volume.subvolume SECURE "security-string";
  ```

- An ALTER TABLE statement for any table whose security or owner is different from the catalog security or owner:

  ```
  >> ALTER TABLE $volume.subvolume.PROGRAMS
  +>   SECURE "security-string"
  +>   OWNER "group-num, user-num" ;
  ```

Purging SQL Objects and Programs (Step 5)

To purge all SQL objects and programs, do these:

1. Log on as the super ID to avoid security restrictions.
2. Remove all SQL objects from the node except the system catalog, the CATALOGS table, and the SQLCI2 program you are using. From SQLCI, enter this command for each catalog except the system catalog:

   ```
   >> PURGE *.*.* FROM CATALOG catalog-name !,
   +> ALLOWERRORS ON;
   ```

Dropping User Catalogs (Step 6)

To drop all catalogs except the system catalog from the node, enter this statement for each catalog while you are logged on as the super ID:

```
>> DROP CATALOG catalog-name;
```

If you get an error while attempting to drop a catalog, use the SQLCI CLEANUP utility, as described later under Purging Detached SQL Objects (Step 8).

Backing Up and Dropping the System Catalog (Step 7)

Before dropping the system catalog, back up any SQL objects it contains and save a copy of the SQLCI2 program to be recompiled on the renamed or renumbered node by following these steps:

1. If your system catalog contains SQL objects, back up the objects you want to save, or move the objects to another catalog. You can use this BACKUP command to save the objects in the system catalog; enter the command at the command interpreter prompt:

   ```
   20> BACKUP *.*.* FROM CATALOG sys-catalog, AUDITED,
         OPEN, LISTALL
   ```

   In the BACKUP command, *sys-catalog* is the name of the volume and subvolume on which your system catalog resides.

2. Use the SQLCI DUP command to save the SQLCI2 program in ZZSQLCI2:

   ```
   >> DUP SQLCI2, ZZSQLCI2, SAVEALL;
   ```

3. Remove the system catalog, including the CATALOGS table and the SQLCI2 program:

   ```
   >>  EXIT
   21> SQLCI
   >> DROP SYSTEM CATALOG sys-catalog;
   ```

   You cannot drop the system catalog while SQLCI2 is running, as it normally is during an SQLCI session in which you have entered other commands. As the example shows, you probably need to end the current SQLCI session and start a new one before entering the DROP SYSTEM CATALOG command. Alternatively, you can exit from SQLCI and enter the command at the command interpreter prompt:

   ```
   >> EXIT
   21> SQLCI DROP SYSTEM CATALOG sys-catalog
   ```

   Be sure to drop all the user catalogs before attempting to drop the system catalog. You cannot drop the system catalog until all entries for user catalogs are deleted from the CATALOGS table.

Purging Detached SQL Objects (Step 8)

Run the Guardian DSAP utility with the SQL option to determine if your node has detached SQL objects:

```
22> DSAP $volume, SQL, DETAIL
```

Then use the SQLCI CLEANUP utility to purge any detached objects:

```
>> CLEANUP $volume.subvolume.object ! ;
```

For a description of the CLEANUP utility, see the *SQL/MP Reference Manual*.

Renaming or Renumbering Your System

Use the SYSGEN utility to rename or renumber your system.

Reinstalling SQL/MP on a Node

Make sure that the TMF subsystem has been brought back up and the appropriate volumes have been enabled. Then re-create the system catalog and initialize SQL/MP by following the instructions in Installing SQL/MP on page 2-2. The two steps are listed briefly here:

1. Create the system catalog:

   ```
   >> CREATE SYSTEM CATALOG catalog-name;
   ```

2. Initialize SQL/MP:

   ```
   >> INITIALIZE SQL;
   ```

Restoring a SQL/MP Database on a Node

1. Re-create the SQL catalogs with the same security and ownership for the catalogs and catalog tables using the SQLCI CREATE utility.
2. Use one of these:
   - Use TMF to recover all audited files except SQL catalog files using the RECOVER utility.
   - Restore the SQL objects from the backup tape or disk files using the RESTORE utility.
3. Restore SQL programs stored as Guardian files from the backup tape or disk files, optionally with recompilation; for an SQL program stored in an OSS file, use the appropriate OSS utility.
4. Make TMF online dumps of all restored audited objects.
5. Verify database consistency.

Re-creating the SQL Catalogs (Step 1)

Use the SQLCI OBEY command to run the OBEY command file, created previously, with the commands to re-create the catalogs with the same security and ownership.

The catalog owner is the user ID executing the CREATE CATALOG statement. Ownership can later be given to another user ID, if necessary, by using the ALTER CATALOG statement.

Note. You may need to create catalogs before recovery, either manually or by specifying the AUTOCREATE CATALOG option of the RESTORE command.

Restoring TMF Audited Files (Step 2)

Use the TMF RECOVER FILES command to recover all audited files except SQL catalog files. The SQL catalog files were created in Step 1 on page 9-37.

For information about recovering audited files using the TMF RECOVER FILES command, see the *TMF Operations and Recovery Guide.*

Restoring SQL Objects (Step 3)

Restoring SQL objects before restoring SQL programs is recommended. You can issue the RESTORE commands at a TACL prompt, but it is probably more efficient to do the RESTORE operation by using an OBEY command file.

To restore the SQL objects on a node to a new location, use the RESTORE utility with the MAP NAMES and CATALOG options (when needed). Issue the RESTORE command on the node renamed or renumbered.

This command restores objects on a node renamed from \source to \target:

```
56> RESTORE $TAPE2, *.*.*,
    AUDITED,
    MAP NAMES (\SOURCE.$VOL1.*.* TO \TARGET.$DATA1.*.*)
    CATALOG ($DATA1.SALES FOR $DATA1.SALES.*,
             $DATA1.ADMIN FOR $DATA1.PERSNL.*,
             $DATA2.INVENT FOR $DATA1.INVENT.*)
```

Restoring SQL Programs (Step 4)

To restore SQL programs stored as Guardian files, specify the tape containing the backed up programs in a RESTORE command. Use the SQLCOMPILE ON option if you want these programs recompiled.

```
58> RESTORE $TAPE1, *.*.*,
    AUDITED
    SQLCOMPILE ON
```

To restore an SQL program stored in an OSS file, use the appropriate OSS utility. For more information, see the *Open System Services Shell and Utilities Reference Manual*.

Making TMF Online Dumps (Step 5)

When you purge SQL objects, the TMF online dumps of the objects are lost. You must make new online dumps of all audited objects restored, including the catalogs.

For information about making online dumps of SQL objects, see the *TMF Operations and Recovery Guide.*

Verifying Database Consistency (Step 6)

Verify database consistency by using the SQLCI VERIFY utility, as explained in Using VERIFY to Detect Invalid Programs on page 10-4.

10 Managing Database Applications

Managing your database includes supporting the operating requirements and access requirements of your application programs and maintaining valid application programs. Providing this support and maintenance can include both performance-related and operational tasks.

Program Validity

Certain DDL statements and utility commands can invalidate a program and mark the program as invalid in the catalog and program file label. When a program is marked as invalid, it is subject to automatic recompilation on subsequent executions, depending on the values of various compile and run time options.

If the file label changes during these DDL operations, the operation completes without marking the program as invalid. The invalid condition of the program, however, is detected at run time.

If a DDL operation causes a program to become invalid, SQL notifies operations in progress by invalidating all opens of the object that changed, causing the next I/O to fail. In many cases SQL can automatically reopen the object and continue processing. In some cases, however, the application must retry the request. For cursor operations, this typically requires a close and reopen of the cursor to reestablish the open. In other cases, you might need to end abnormally and rerun the current transaction. If you obtain an open invalidation error, see the associated error message for specific recovery information.

Operations That Invalidate a Program

- Copying the program file. If you copy a program file by using the FUP DUPLICATE command, the original file is unaffected, but the new file is invalid. For more information, see Moving Programs on page 10-39.
- Binding the program file. If you explicitly bind a program file by using the Binder program, the original file is unaffected, but the resulting target file is invalid.
- Restoring a program file. If you restore a program file (or an underlying table of a protection or shorthand view used by the program) by using the RESTORE program without specifying the SQLCOMPILE ON option, the restored program is invalid.
- Running the Accelerator for the program file. If you run the accelerator to optimize the object code (TNS/R systems only), the program file becomes invalid.

These changes to SQL objects used by an SQL program file invalidate the program file:

- Adding a constraint to a table used by the program
- Adding a column or partition to a table used by the program (including an underlying table of a protection or shorthand view used by the program) unless the program is compiled with the CHECK INOPERABLE PLANS option and the table and any associated protection views have the similarity check enabled. (For more information about similarity checks, see Using Similarity Checks on page 10-15.)
- Adding an index to a table used by the program, or to an underlying table of a protection or shorthand view used by the program, unless you specify the NO INVALIDATE option in the CREATE INDEX statement or unless the program is compiled with the CHECK INOPERABLE PLANS option and the table and any associated protection views have the similarity check enabled.
- Changing a collation: dropping and then re-creating the collation, renaming a collation, or changing a DEFINE that points to a collation
- Executing the UPDATE STATISTICS statement unless you specify the NORECOMPILE option on tables used by the program or unless the program is compiled with the CHECK INOPERABLE PLANS option and the table and protection views referenced by the program have the similarity check enabled.
- Dropping or doing a cleanup on a table or view
- Dropping a partition of a table or index unless the program is compiled with the CHECK INOPERABLE PLANS option and the table referenced by the program has the similarity check enabled
- Dropping an index or constraint on a table
- Restoring a table, including an underlying table of a protection or shorthand view, using the RESTORE program
- Changing the PARTITION ARRAY type associated with the base table

To maintain valid programs, you need procedures that explicitly SQL compile affected programs after these listed operations occur. Otherwise, automatic recompilation occurs at run time. For more information, see Explicit Compilation on page 10-6 and Automatic Recompilation on page 10-7.

Unexpected Events That Can Invalidate a Program

Sometimes, object program files are created and appear to be valid but are not. These events can produce such a situation:

- After an invalidating DDL change if the program's catalog was available during the change but the object program file was not available
- When a processor failure or other event destroys the SQL compiler process and its context after the compiler has produced the object file, has updated the SQL

catalog to register the program, and has marked the object file label as SQL-sensitive and valid, but before the compiler normally terminates

If a compilation terminates abnormally, the TMF subsystem backs out the updates to the catalog but cannot undo the changes to the object file label because the label for an SQL object file is always nonaudited. In such a case, a seemingly valid object file exists on disk, but no entry for this file exists in the PROGRAMS table of the catalog.

You can sometimes recover from this condition by running SQLCOMP again to reenter the information in the catalog. If this strategy does not resolve the problem, use the CLEANUP utility or the GOAWAY utility to remove the object file, and recompile the program. (For more information about using the CLEANUP and GOAWAY utilities, see the *SQL/MP Reference Manual*.)

Operations That Do Not Invalidate a Program

Not all changes to the program or database invalidate a program. These operations do not invalidate a program:

- Altering the security or owner of the program or SQL objects
- Creating new views on a table
- Altering file attributes, including the AUDIT flag
- Adding or dropping comments on a table or view
- Adding a column or partition if the CHECK INOPERABLE PLANS option is used and referenced tables and protection views have the similarity check enabled
- Adding an index with the NO INVALIDATE option in the CREATE INDEX statement or if the CHECK INOPERABLE PLANS option is used and referenced tables and protection views have the similarity check enabled
- For the source object, duplicating a program or SQL object
- Executing the UPDATE STATISTICS statement with the NORECOMPILE option on tables used by the program; however, the new statistics might enable the SQL compiler to determine a better access path for the programs.

Although changing the AUDIT attribute of a table referred to by an SQL statement does not invalidate the statement, this change does cause automatic SQL recompilation (if it is specifically allowed) in these cases:

- If a statement performs a DELETE or UPDATE set operation on a nonaudited table that has a SYNCDEPTH of 1, the SQL executor returns SQL error 8203 and forces automatic recompilation of the statement.
- If a statement is executed in parallel on a table whose AUDIT attribute has changed since the last explicit SQL compilation, the SQL executor returns SQL error 8207 and forces automatic recompilation of the statement.

Determining Validity of a Program

A program is invalid if any of these are true:

- The VALID flag of the program entry in the PROGRAMS table is not set or is set to *N*.
- The VALID flag in the program file label is not set or does not correspond to the VALID flag in the PROGRAMS entry.
- The value of RECOMPILETIME in the program file label does not correspond to the RECOMPILETIME recorded in the PROGRAMS catalog table.
- The RECOMPILATION timestamp in the program file label represents a time earlier than any redefinition timestamp of any SQL object on which the program depends.

There are several ways to verify and maintain valid programs in your application:

- Use the VERIFY utility through SQLCI to read the PROGRAMS catalog tables and the program file labels to determine validity.
- Query the PROGRAMS catalog table directly to find programs marked as invalid.
- Monitor SQL compilations and automatic recompilations by using the logging facility.

Using VERIFY to Detect Invalid Programs

You can use VERIFY to check file labels and catalogs for invalid programs. VERIFY can produce an output EDIT file that contains a list of the invalid programs. You can edit the invalid warning message for each invalid program written to an EDIT file to create an OBEY command file to explicitly SQL compile these programs.

VERIFY can detect only those programs actually marked as invalid in the file labels and catalogs tables. VERIFY cannot detect all the conditions that could cause automatic recompilation at run time.

VERIFY does not detect the invalid status of programs in these situations:

- The FORCE option was used.
- A program was explicitly SQL compiled, but the best query execution plan was not available at compile time.

In the latter case, the valid flag is set to Y, but certain statements in the program are invalid and must be automatically recompiled at run time.

To maintain valid programs, use VERIFY in combination with the logging facility that detects all automatic recompilations. This VERIFY example shows the invalid warning messages. A VERIFY request generates SQLCOMP commands to recompile any invalid programs. In this example, these commands are written to a cleared EDIT file named COMPFILE. You can edit this file and use it as a command file in an OBEY command, directing the command interpreter to recompile the programs.

```
>>  VERIFY *.*.* WHERE SQLPROGRAM, SOURCE COMPFILE CLEAR;
...
--- Verifying  $VOL1.PPROGS.UEMPLIST
*** WARNING  $VOL1.PPROGS.UEMPLIST IS AN INVALID PROGRAM.

--- SQL operation complete.
```

Querying the PROGRAMS Catalog Table

You can query the catalog tables to verify whether VALID flags are set. You can check the catalog VALID flags but not the program's file label. Similar to the VERIFY utility, these queries might not detect all conditions that can cause programs to be automatically recompiled.

The PROGRAMS table of the catalog stores information about program validity. You can query each catalog for invalid programs. If you set up an SQLCI log file, the output of the query is duplicated in an EDIT file.

This example shows setting a log file and then querying the PROGRAMS table:

```
>>  LOG $SYSTEM.PGMS.INVALID;
>>  CATALOG \SYS1.$VOL1.SALES;
>>  SELECT * FROM PROGRAMS
+>     WHERE VALID = "N" OR
+>           AUTOCOMPILE = "Y";
```

△ **Caution.** If a program is marked as invalid in the catalog or is detected as invalid by the VERIFY utility, you should explicitly SQL compile the program to revalidate the program to avoid automatic recompilations.

You should not attempt to validate a program file by altering the VALID column in the PROGRAMS table. Validation information is also stored in the program's file label, which cannot be altered with SQL utilities.

Monitoring Compilations

If you want to be sure that all programs are valid, you should monitor recompilation with the SQL logging facility. The logging of messages about explicit compilations and automatic recompilations (described later in this section) is automatically directed to $0 for certain SQL compilation events. Compilations can be initiated by SQLCI commands (which use dynamic SQL), dynamic SQL statements, explicit SQL compiles, or automatic SQL recompilation.

You can use the DEFINE =_SQL_CMP_EVENT to redirect the logging to a disk file, or a terminal, or to disable the logging facility. Logging to $0 is automatic unless you disable the logging. Exceptionally heavy SQLCI DML activity or compilation activity can exceed the capacity of $0. For the description of how to set up and use the DEFINE =_SQL_CMP_EVENT to control the device to which messages are logged, see the *NonStop SQL/MP Reference Manual*.

Logging might be especially helpful on a system where automatic recompilations are not wanted for performance reasons. Examine any program that has a recompilation logged to determine whether the program needs explicit recompilation. Use VERIFY to check the program entry in the catalog.

Programs marked invalid in the catalog need explicit SQL compilation. A program or statement could be logged for recompilation but might not be invalid, such as a recompilation because of an unavailable node. Programs recompiled at run time but not otherwise invalid do not need explicit compilation.

SQL Compilation and Recompilation

SQL compilation verifies the use of SQL objects, optimizes access paths to the database for each SQL statement, and writes the object code for the plan to use the chosen paths. Successful SQL compilation always generates an executable query execution plan for each SQL statement in all interfaces, namely explicit SQL compilation, automatic recompilation, dynamic SQL statements, and SQLCI ad hoc queries (which are dynamic SQL statements).

The results of a compilation depend on whether the compilation is explicit or automatic and on the SQL compiler options that are in effect.

Results also depend on statistics. SQL compilation uses statistics in the catalogs to determine access paths, depending on the availability of the objects. Unavailable objects, such as an index, affect the path chosen. For more information about statistics, see Keeping Statistics Current on page 14-7.

This subsection describes features of SQL compilation, including explicit compilation and automatic recompilation, SQL compiler options that control recompilation, and query execution plans. For additional information about query execution plans, see the *SQL/MP Query Guide*.

Explicit Compilation

Explicit compilation occurs when you run the SQL compiler, specifying a host-language object program file. Explicit compilation also occurs for SQL programs stored in Guardian files when you specify the SQLCOMPILE ON option for the RESTORE utility. The results of successful explicit compilation are:

- Executable object code is generated in the program file for the optimized access paths.
- The file label of the program file is marked with the SQL SENSITIVE and SQL VALID flags being true (set on).

- The program file is registered in the catalog. This operation includes storing a description of the program in the PROGRAMS catalog table and storing usage dependencies in the USAGES tables of the catalogs in which objects referred to by this program are described.
- Explicit compilation produces an object program that can be executed without first being automatically recompiled. To avoid the overhead of automatic recompilation, you must ensure that programs are valid as described under Determining Validity of a Program on page 10-4.
- If similarity checking is enabled by using the CHECK INOPERABLE PLANS option, certain types of run-time recompilations can be minimized or avoided (described in the next subsection).

Automatic Recompilation

Automatic recompilation is the SQL recompilation, in memory, of a program or SQL statement. The recompilation is invoked automatically by the SQL executor at run time.

The RECOMPILE option (described later in this subsection) is required during the explicit SQL compilation if you want to enable subsequent automatic recompilation for the program. The extent of recompilation depends on whether the RECOMPILEALL or RECOMPILEONDEMAND compiler option is used for explicit compilation.

Automatic recompilation can occur in these situations:

- A program file is marked as invalid. The SQL compiler CHECK option (described under Using Similarity Checks on page 10-15) determines the extent of recompilation. Similarity checking can avoid or minimize recompilation.
- DEFINEs at run time are different from the values of the DEFINEs in effect at explicit compilation time. The SQL compiler CHECK option determines the extent of recompilation.
- An event occurs during execution that changes the definition of an object used by a program. (The executor determines that the RECOMPILATION timestamp in the program file label represents a time that is earlier than the redefinition timestamp of a dependent SQL object.)
- Some objects required for the query execution plan are not available at run time. The objects could be local or remote tables, views, or indexes.
- RECOMPILEONDEMAND is selected, and an attempt is made to run an invalid statement.
- The access path used by the plan is not available. If an index is not available, the executor can recompile the plan to use the primary access path.
- The program was compiled with the FORCE option, and some statements had errors; the statements with errors are automatically recompiled.

- The AUDIT attribute of a table referred to by an SQL statement is altered. This does not invalidate the statement, but in these cases altering the AUDIT attribute can cause automatic recompilation:
    - If a statement performs a DELETE or UPDATE set operation on a nonaudited table with a SYNCDEPTH of 1, the SQL executor returns SQL error 8203 and forces automatic recompilation of the statement.
    - If a statement is executed in parallel on a table whose AUDIT attribute has changed since the last explicit SQL compilation, the SQL executor returns SQL error 8207 and forces automatic recompilation of the statement.

The SQL executor detects the condition requiring recompilation and invokes the SQL compiler. The compiler generates an execution plan based on the available information and the best available access path.

If a statement cannot be executed again because of another invalid path, a last attempt is made to compile the statement by using the primary key as the access path. The SQL executor tries to recompile only two times. If, in these two attempts, the SQL compiler cannot find an available access path that returns all requested data, the data is considered unavailable, and an error is returned to the program for the statement.

Subsequent use of the same statement within the same process (SQLCI session) uses the plan developed by the previous recompilation, so that recompilation does not occur again.

If an access path becomes unavailable before the execution of an SQL statement finishes, the SQL executor takes one of these actions:

- If no records were returned during the statement execution, the SQL executor again attempts recompilation as described in the preceding paragraphs.
- If one or more records were returned during the statement execution, the SQL executor returns an error indication and terminates.

Automatic recompilation does not occur if an object that was initially unavailable becomes available while a program is running. If a program was automatically recompiled because a path in the best query execution plan became unavailable, the program continues to run with the best available access path until the program is stopped and restarted.

The results of automatic recompilation are:

- Executable object code for the optimized access paths is generated in the copy of the program file in the SQL executor's memory.
- The program is validated only for the duration of the current session. No compilation changes or validation flags are stored in the program file or in catalogs.

You can reduce compilation time for an application by directing the SQL compiler to recompile only plans that are actually inoperable, not merely invalid. If you do so, the SQL compiler uses similarity checks to determine whether certain invalid plans (those that are invalid because objects they reference have been changed or redefined) are actually operable plans.

Note. An operable plan, while executable, might not be optimal—and that recompilation might improve performance. For more information, see Using Similarity Checks on page 10-15.

Performance Considerations

Performance is best if you ensure that programs do not need recompilation at run time. Some programs can be marked valid and still require automatic recompilation, so plan to eliminate as many causes of automatic recompilation as possible, as described under Program Validity on page 10-1. Options are available that govern the extent of recompilation for an application; these options are discussed in this subsection.

Monitor programs that are automatically recompiled because of unavailable objects so that you can restart the programs when the object becomes available. If you are running applications in a Pathway environment, for example, you can stop and restart the server class to initiate use of the program version with the best access path. For other programs, stop and restart the programs.

SQL Compiler Options for Recompilation

These compiler options control whether recompilation occurs:

- RECOMPILE specifies that if the program becomes invalid, the system is to recompile either the whole program or the SQL statements used, depending on the choice of the RECOMPILEALL or RECOMPILEONDEMAND option. RECOMPILE sets the AUTOCOMPILE flag in the PROGRAMS table to *Y*. This option is the default.

  For continuous access to your database or for local autonomy in a distributed database, use RECOMPILE to ensure:

  - Programs run if access paths are available although database changes occurred since the programs were explicitly compiled.
  - You have local autonomy because the system can automatically recompile programs to determine access paths if access paths through other systems become unavailable.

- NORECOMPILE specifies that if the program becomes invalid, the system cannot automatically recompile it. The resulting program file cannot run if the program becomes invalid or if tables or indexes become unavailable. NORECOMPILE sets the AUTOCOMPILE flag in the PROGRAMS table to *N*.

  To control recompilations by detecting the occurrence of an invalid program immediately, use NORECOMPILE. This option ensures that any cause of invalidity is immediately detected, because an invalid program cannot run. With

NORECOMPILE, the only way to revalidate the program is to explicitly SQL compile it again. By using NORECOMPILE, you ensure that:

- All programs use the best query execution plan to enhance performance.
- Alterations to the database, object moves, or errors in the DEFINE setup cannot inadvertently affect the production environment.

If you specify RECOMPILE, determine the extent of recompilation that is best for your application. Use the SQL compiler RECOMPILEALL or RECOMPILEONDEMAND option that causes the least overhead for the recompilations:

- RECOMPILEALL specifies that if the program becomes invalid, the entire program is automatically recompiled before execution continues. After being automatically recompiled, the program is executed as any valid program. This option is the default.
- RECOMPILEONDEMAND specifies that if the program becomes invalid, only the statements actually executed are automatically recompiled. Recompilation occurs on the first execution of each invalid statement. This option can save system resources if the server has a number of unexecuted SQL statements.

PATHMON DEFINEs and SQL Recompilation

Setting DEFINE names in the PATHMON configuration file does not change DEFINEs that were in effect when a server was SQL compiled.

After a program has been SQL compiled using a specified DEFINE set, the program is valid only for those objects in that set. Changing the DEFINE set in the PATHMON configuration file triggers an automatic recompilation at run time if the DEFINE set is different from that used at compilation time.

If no DEFINEs are set within the configuration file for the server class, the server uses the DEFINE set that was valid during the last explicit SQL compilation.

While it is always advantageous to run valid SQL programs (programs that do not recompile at run time), setting the DEFINE set within the configuration ensures that all programs use the proper DEFINE set, regardless of the last explicit SQL compilation. The DEFINE set within the PATHMON configuration for the server class and the DEFINE set used at compilation should be identical.

If an SQL program is explicitly SQL recompiled while the previously generated object program is running, a ZZBI*nnnn* file is produced. The object program currently in use is renamed ZZBI*nnnn*, and the newly compiled program becomes the named program. The PATHMON process continues to use the ZZBI*nnnn* program for the servers that are presently in run state. New server programs started by the PATHMON process, however, will use the newly compiled program.

If the object program is a native application (code 700), the currently in use program file is renamed as ZZND*nnnn* instead of ZZBI*nnnn*. Here *nnnn* denotes a random four digit number.

Following the explicit recompilation, you can stop and start the server class to ensure that all running objects use the newly compiled program. These commands illustrate this action:

```
= FREEZE SERVER SRV-SDB102
= STOP SERVER SRV-SDB102
= THAW SERVER SRV-SDB102
= START SERVER SRV-SDB102
```

For more information about DEFINEs see Using DEFINEs on page 10-30.

For information about execution-time name resolution and how it relates to PATHMON DEFINES, see Deferring Name Resolution on page 10-13.

Explicit Compilation and Query Execution Plans

A query execution plan is an execution method, including the semantics and execution characteristics, for a compiled SQL statement. The SQL compiler stores the execution plan in the program file during explicit compilation, and the SQL executor uses the plan to run the SQL statement at run time.

For queries, the SQL optimizer usually examines as many execution plans as there are ways to access the data. The optimizer then chooses the plan it considers the most efficient, based on the number of input-output operations and the use of processor time. The plans include any indexes used and any sort operations performed to order the data as requested. The EXPLAIN utility displays the query execution plan chosen by the optimizer.

The SQL compiler tries to compile each DML statement with the best query execution plan, which provides the best performance. The compiler can determine the best query execution plan only when all the required information for the SQL objects referred to in the statements is available and current.

Necessary information includes:

- The catalogs of the referenced objects. These catalogs contain the description of the objects. The referenced objects can be local or remote tables or views and local or remote partitions of tables.
- The statistics on the tables and associated indexes.

Both explicit compilation and automatic recompilation determine query execution plans for DML statements. The types of query execution plans depend on the type of SQL compilation.

Explicit SQL compilation requires all the necessary information to be available to create the best query execution plan. If some of the required information is not available, the compiler cannot generate a query execution plan for the affected statements.

If the compilation is otherwise successful in generating an object file, the program is registered and marked valid in the catalog; the statements for which the best query execution plan could not be generated are marked invalid on a statement-by-statement basis. At run time, these statements are automatically recompiled when the compiler directives allow this recompilation.

The quality of the query execution plans depends on the accuracy of the statistics used by the compiler when determining the plan. If the statistics are not current, compilation can cause a valid program, but the chosen query execution plan could give less than the best performance. For the best performance for programs, you must ensure that the statistics represent the current state of the table and indexes with reasonable accuracy.

If you want to review the chosen query execution plans, you can SQL compile the program and use the EXPLAIN option. The EXPLAIN utility reports the access paths for each DML statement and, optionally, the DEFINEs in effect. For additional information about the SQL compiler EXPLAIN utility, see the *SQL/MP Reference Manual*.

Automatic Recompilation and Query Execution Plans

For automatic recompilation of programs or statements or for SQLCI queries or dynamic SQL queries, the SQL compiler tries to generate the best query execution plan that provides the best performance. If some of the required information is not available, the compiler tries to generate the best available query execution plan using the available objects.

These are points to consider about the best available query execution plan:

- The best available query execution plan is the optimal access plan if all objects are available. This best query execution plan typically results from automatic recompilations because of these conditions:
    - The program was previously explicitly compiled, but statements were left uncompiled because objects were not available.
    - The program was automatically recompiled because DDL operations on referenced tables invalidated the program.
- The best available query execution plan generated by automatic recompilation is typically not the optimal plan when objects required for the previously compiled best query execution plan are not available. In such a case, the best available query execution plan can produce suboptimal performance. The plan, however, ensures access to the data.

For automatic recompilation, the file label of the named or underlying table and the catalog in which the named object is registered must be available. For a partitioned table, only the file label of the specified partition must be available.

For statement execution, any partition of the named object can be opened. Other partitions must be available only if the query requests data from another partition and the SKIP UNAVAILABLE PARTITION option is not in effect.

For the execution of an INSERT or DELETE statement, all affected partitions of the table and corresponding partitions of all indexes must be available. For the execution of an UPDATE statement, all affected partitions of the table and only the affected partitions of indexes that include columns being updated must be available.

Consider this example:

```
Table X (columns A1, B1, C1, C2, C3, C4) resides on $VOL1
Index A using column A1 and B1 resides on $VOL3
Index B using column A1 and C4 resides on $VOL1
```

If a query is compiled to use index A as the access path and $VOL3 is down, the query is recompiled to attempt to get the data by using index B or by using the primary key. If you attempt to insert a row into table X with values for all the columns, the insert fails if $VOL3 is unavailable. If you attempt an update for table X, in which columns C1 and C2 are updated, the update completes even if $VOL3 is unavailable. Index A, which resides on $VOL3, is not required for the update.

Deferring Name Resolution

Execution-time name resolution is the resolution of the name of an SQL object (table, view, index, or partition), program, or catalog in an SQL statement, at statement execution time rather than during explicit SQL compilation or at SQL load time. Thus, a program can resolve SQL names at statement execution time without using dynamic SQL statements. You can develop programs that run SQL statements against different tables than those for which the programs were originally compiled.

SQL/MP resolves SQL names in static SQL statements as described next and as shown in Figure 10-1 on page 10-14:

- During explicit SQL compilation, SQL/MP resolves the names in SQL statements.
- At SQL load time (which is when the first SQL statement in the program executes), the SQL executor resolves the SQL names again, if the program is invalid or a DEFINE specified in an SQL statement has changed since the last explicit SQL compilation.
- If execution-time name resolution is enabled, the SQL executor resolves names in an SQL statement when the statement actually executes.

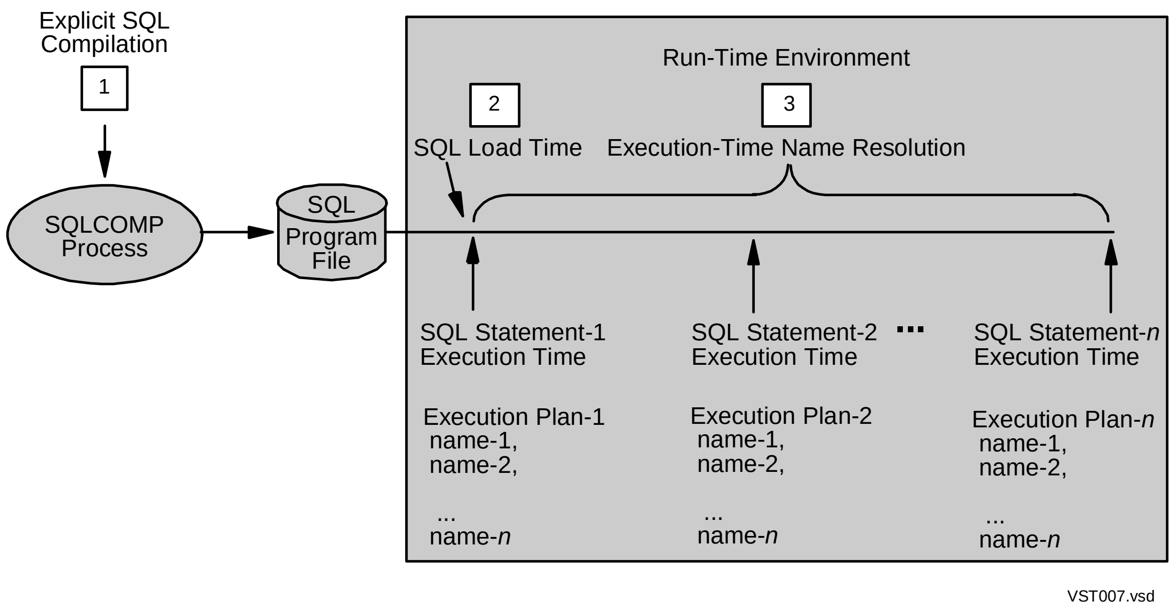
Execution-time name resolution applies to Guardian names and DEFINE names as follows:

Name	**Description**
Guardian file name Class MAP DEFINE	SQL object (table, view, collation, index, and partition), or SQL program
Guardian subvolume name Class CATALOG DEFINE	SQL catalog

Execution-time name resolution is implemented by an option in the CONTROL QUERY statement.

The sample database described in the *SQL/MP Reference Manual* includes a sample program that uses execution-time name resolution. For sample scenarios of use, see Using DEFINEs on page 10-30.

Figure 10-1. Name Resolution For SQL Statements


Explicit SQL Compilation
1
SQLCOMP Process
SQL Program File
Run-Time Environment
2
SQL Load Time
3
Execution-Time Name Resolution
SQL Statement-1 Execution Time
SQL Statement-2 Execution Time
...
SQL Statement-*n* Execution Time
Execution Plan-1 name-1, name-2, ... name-*n*
Execution Plan-2 name-1, name-2, ... name-*n*
Execution Plan-*n* name-1, name-2, ... name-*n*
VST007.vsd

CONTROL QUERY BIND NAMES Directive

Execution-time name resolution is requested by using the CONTROL QUERY BIND NAMES directive. This directive is used at the statement level in SQL programs and, as such, is not accessible at a system management level. Using this option, however, might be appropriate for your application. For more information about the CONTROL QUERY BIND NAMES directive, see the SQL/MP programming manual for your host language.

Avoiding Automatic SQL Recompilations

To prevent unnecessary automatic SQL recompilations of a program or SQL statement, use the similarity check with execution-time name resolution. You use DDL statements to enable the similarity check for each table or protection view referenced in the statement. (SQL implicitly enables the similarity check for other SQL objects.) To enable the similarity check for an SQL program, compile the program by using the CHECK INOPERABLE PLANS option. For more information, see Using Similarity Checks on page 10-15.

Using Similarity Checks

A similarity check is a comparison made by SQL to determine whether two objects (or the compile-time and execution time version of the same object) are sufficiently similar that a serial execution plan compiled for one can work as an operable plan for the other. For example, if a statement refers to a table at run time, which is similar to the table the statement was compiled against, SQL/MP allows the statement to run without automatic recompilation.

Similarity checks work by comparing information stored in an execution plan with information current at recompilation time. Executing the similarity check is faster than recompiling an execution plan, can potentially avoid a recompilation, and can therefore reduce the downtime for an SQL program.

The similarity check is done on a per-statement and per-object basis. There are three aspects of setting up similarity checking:

- Specifying similarity checking at compile-time of a program
- Specifying similarity checking at run time of the program
- Enabling similarity checking for specific tables and collations accessed by the program; it is explicitly enabled for all other objects except shorthand views

Because SQL/MP must have information about programs, objects, and views before doing similarity checking, all three actions are necessary.

You can use similarity checking along with execution-time name resolution or to recover from DDL compilations; in both cases you can avoid automatic recompilations.

This subsection includes this information about similarity checking:

- Using the CHECK option to direct the SQL executor to perform the similarity check at execution time for recompilations
- Using the COMPILE option to direct the SQL compiler to perform the similarity check during explicit SQL compilation
- Enabling the similarity check for tables and protection views using DDL statements
- Enabling similarity checking for table and protection views

Using the CHECK Option

The CHECK option determines the behavior of the SQL executor at run time, during an automatic recompilation, when it executes an invalid SQL statement or a statement that refers to a DEFINE that has changed since the last explicit SQL compilation.

You can direct the SQL executor to use the similarity check to determine if a statement’s execution plan is operable and can run without automatic recompilation. The SQL executor then recompiles only the SQL statements that fail the similarity check; it executes other SQL statements using their existing plans.

The CHECK option has three forms:

- INVALID PROGRAM specifies automatic recompilation for all SQL statements in an invalid program, or a program that refers to changed DEFINEs (if NORECOMPILE is not specified). This option is the default.
- INVALID PLANS specifies automatic recompilation for an SQL statement if either of these conditions occur (and NORECOMPILE is not specified):
    - The statement is invalid.
    - The statement refers to a DEFINE at SQL load time that has changed since the last explicit SQL compilation.
- INOPERABLE PLANS specifies that the SQL executor should perform the similarity check for each SQL object in an SQL statement if the similarity check is enabled for referenced tables and protection views and either of these conditions occur:
    - The statement is invalid.
    - The statement refers to a DEFINE at SQL load time that has changed since the last explicit SQL compilation.

  If the similarity check passes, the SQL executor considers the plan to be operable (although it might not be optimal) and executes the statement without automatically recompiling it.

  If the similarity check fails, the SQL executor considers the plan to be inoperable. The SQL executor then recompiles (in memory only) the statement that generated the inoperable plan (if NORECOMPILE is not specified) and executes the recompiled statement.

Parallel Execution Plans

You cannot use the similarity check for a query that uses parallel execution plans. At run time, a query that uses parallel execution plans fails the similarity check, and the SQL statement containing the query must be automatically recompiled before it can run (if NORECOMPILE is not specified). To use the similarity check in this query, you must disable parallel plans by using a CONTROL QUERY PARALLEL EXECUTION OFF directive.

Preventing Program Invalidation Caused by DDL Operations

Certain DDL operations on an SQL object cause a program that refers to the object to be invalidated. When a program is invalidated, the SQL catalog manager sets the VALID flag to *N* in the PROGRAMS catalog table and in the program’s file label (if the program file is accessible) and deletes the program’s usages entries in the USAGES table. An invalid program must be recompiled either explicitly or automatically before it can execute.

These DDL operations do not invalidate a program compiled with the CHECK INOPERABLE PLANS option if the similarity check is enabled for each referenced object. The program also retains its entries in the USAGES table. (These operations, however, do update the redefinition timestamp of each referenced object in the DDL statement.)

- ALTER TABLE...ADD PARTITION statement
- ALTER TABLE...ADD COLUMN statement (for more information, including restrictions, see Enabling the Similarity Check for Tables and Protection Views on page 10-26)
- ALTER TABLE statement to move or split partitions (including a simple move, one-way split, or two-way split) or change the type of partition array
- ALTER TABLE...DROP PARTITION statement
- ALTER INDEX...DROP PARTITION statement (if the similarity check is enabled for the base table)
- ALTER INDEX statement to move or split index partitions
- CREATE INDEX statement
- UPDATE STATISTICS...RECOMPILE statement

The ALTER TABLE... RENAME, ALTER INDEX... RENAME, and ALTER INDEX... ADD PARTITION statements do not invalidate a program whether or not it was compiled with the CHECK INOPERABLE PLANS option.

Note. These DDL operations always invalidate a program, even if the program is compiled with the CHECK INOPERABLE PLANS option:

- ADD CONSTRAINT statement
- DROP CONSTRAINT statement
- DROP TABLE statement
- DROP VIEW statement
- ALTER TABLE or ALTER VIEW statement with the SIMILARITY CHECK clause (for more information, see Enabling the Similarity Check for Tables and Protection Views on page 10-26)
- DROP INDEX statement if the program contains a plan that refers to the dropped index

Example: Preventing Recompilations After a DDL Change

To prevent recompilation, enable the similarity check for all referenced tables and protection views and compile the program with the CHECK INOPERABLE PLANS option. To do this, follow these steps:

1. Enable the similarity check for each table or protection view specified in the SQL statements as follows:
    - For existing tables, use the ALTER TABLE or ALTER VIEW statement with the SIMILARITY CHECK ENABLE clause.
    - If you are creating a new table or protection view, use the CREATE TABLE or CREATE VIEW statement with the SIMILARITY CHECK ENABLE clause.
2. Explicitly SQL compile the program with the CHECK INOPERABLE PLANS option to enable the similarity check.
3. Run the program as usual. These DDL operations do not invalidate the program, because it was compiled with the CHECK INOPERABLE PLANS option and uses the similarity check for any referenced tables or protection views:
    - ALTER TABLE...ADD PARTITION statement
    - ALTER TABLE...ADD COLUMN statement (for more information, including restrictions, see Enabling the Similarity Check for Tables and Protection Views on page 10-26)
    - ALTER TABLE statement to move or split partitions (including a simple move, one-way split, or two-way split)
    - ALTER TABLE...DROP PARTITION statement
    - ALTER INDEX...DROP PARTITION statement
    - ALTER INDEX statement to move or split index partitions
    - CREATE INDEX statement
    - UPDATE STATISTICS...RECOMPILE statement

    Also, if a DDL operation does cause a program to be invalidated, the SQL executor still performs the similarity check. If the similarity check passes for an SQL statement, the SQL executor executes the statement without recompiling it.

Interaction Between the CHECK Option and Other SQLCOMP Options

Table 10-1 on page 10-20 describes the actions of the SQL executor when it runs an SQL program compiled with a CHECK option and the RECOMPILE, NORECOMPILE, RECOMPILEONDEMAND, or RECOMPILEALL option for this situation.

At SQL load time, the SQL executor detects invalid statements or statements that refer to a DEFINE that has changed since the last explicit SQL compilation. (SQL load time occurs when a program executes its first SQL statement.)

Table 10-1. Behavior of the SQL Executor for an Invalid Statement or a Changed DEFINE Detected at SQL Load Time (page 1 of 2)

SQLCOMP Options		**Behavior**
CHECK INVALID PROGRAM Option		
RECOMPILE	RECOMPILEALL	The SQL executor recompiles (in memory) all SQL statements. This option is the default behavior.
RECOMPILE	RECOMPILEONDEMAND	The SQL executor recompiles (in memory) a statement the first time it is executed using Guardian names and DEFINE names as follows: ● Uses names at SQL load-time if execution-time name resolution is not enabled ● Uses names at statement execution time if execution-time name resolution is enabled
NORECOMPILE	RECOMPILEALL or RECOMPILEONDEMAND	The SQL executor returns an error to the program.
CHECK INVALID PLANS Option		
RECOMPILE	RECOMPILEALL	The SQL executor recompiles (in memory) only invalid SQL statements and statements that refer to changed DEFINEs. The SQL executor executes other statements using existing plans.
RECOMPILE	RECOMPILEONDEMAND	The SQL executor recompiles (in memory) the statement the first time it is executed using Guardian names and DEFINE names as follows: ● Uses names at SQL load time if execution-time name resolution is not enabled ● Uses names at statement execution time if execution-time name resolution is enabled The SQL executor executes other statements using existing plans from the program file
NORECOMPILE	RECOMPILEALL or RECOMPILEONDEMAND	The SQL executor returns an error to the program.

Table 10-1. Behavior of the SQL Executor for an Invalid Statement or a Changed DEFINE Detected at SQL Load Time (page 2 of 2)

SQLCOMP Options		**Behavior**
CHECK INOPERABLE PLANS Option		
RECOMPILE	RECOMPILEALL	The SQL executor performs the similarity check as follows: ● If the similarity check passes, the SQL executor executes the statement using its existing plan from the program file. ● If the similarity check fails, the SQL executor recompiles (in memory) the statement at SQL load time.
	RECOMPILEONDEMAND	The SQL executor performs the similarity check for the statement the first time it is executed as follows: ● If the similarity check passes, the SQL executor executes the statement using existing plans from the program file. ● If the similarity check fails, the SQL executor recompiles (in memory) the statement using Guardian names and DEFINE names as follows: Uses names at SQL load time if execution-time name resolution is not enabled Uses names at statement execution time if execution-time name resolution is enabled
NORECOMPILE	RECOMPILEALL or RECOMPILEONDEMAND	The SQL executor performs the similarity check for the statement as follows: ● If the similarity check passes, the SQL executor executes the statement using plans from the program file. ● If the similarity check fails, the SQL executor returns an error to the program because of the NORECOMPILE option.

Table 10-2 describes the behavior of the SQL executor when it encounters an invalid static or dynamic statement during the execution of an SQL program compiled with a CHECK option and the RECOMPILE or NORECOMPILE option for this situation. A statement is invalidated during program execution when a DDL operation takes place during program execution.

After SQL load time, the SQL executor detects invalid static or dynamic SQL statements. (The RECOMPILEALL and RECOMPILEONDEMAND options do not apply to this situation.)

Table 10-2. Behavior of the SQL Executor for an Invalid Statement Detected After Load Time (page 1 of 2)

SQLCOMP Option	Behavior
CHECK INVALID PROGRAM or CHECK INVALID PLANS Option	
RECOMPILE	The SQL executor recompiles (in memory) the invalid SQL statement using Guardian names and DEFINE names as follows: ● Uses names at SQL load time if execution-time name resolution is not enabled ● Uses names at statement execution time if execution-time name resolution is enabled
NORECOMPILE	The behavior depends on the type of statement: ● For static SQL statements, the SQL executor returns an error to the program and does not recompile the statement because of the NORECOMPILE option. ● For dynamic SQL statements, the SQL executor recompiles the statement. The NORECOMPILE option has no effect on dynamic SQL statements, and invalid dynamic plans are always recompiled.
CHECK INOPERABLE PLANS Option	
RECOMPILE	For an invalid static SQL statement, the SQL xecutor performs the similarity check as follows: ● If the similarity check passes, the SQL executor executes the statement without recompilation. ● If the similarity check fails, the SQL executor recompiles (in memory) and executes the statement using Guardian names and DEFINE names as follows: Uses names at SQL load time if execution-time name resolution is not enabled Uses names at statement execution time if execution-time name resolution is enabled

Table 10-2. Behavior of the SQL Executor for an Invalid Statement Detected After Load Time (page 2 of 2)

SQLCOMP Option	Behavior
CHECK INOPERABLE PLANS Option (continued)	
RECOMPILE (continued)	For an invalid dynamic SQL statement, the SQL executor performs the similarity check as follows: ● If the similarity check passes, the SQL executor executes the statement without recompilation. ● If the similarity check fails, the SQL executor recompiles (in memory) and executes the statement using Guardian names and DEFINE names as follows: Uses names at prepare time if execution-time name resolution is not enabled Uses names at statement execution time if execution-time name resolution is enabled
NORECOMPILE	The SQL executor performs the similarity check for the invalid SQL statement as follows: ● If the similarity check passes, the SQL executor executes the statement without recompilation. ● If the similarity check fails, the behavior depends on the type of statement: ● For a static SQL statement, the SQL executor returns an error to the program because of the NORECOMPILE option. ● For a dynamic SQL statement, the SQL executor recompiles the statement because the NORECOMPILE option has no effect on dynamic SQL statements. Invalid dynamic plans are always recompiled.

Using the COMPILE Option

The COMPILE option influences the behavior of the SQL compiler. The option determines which SQL statements are compiled during an explicit SQL compilation. You can direct the SQL compiler to use the similarity check to determine if a statement's execution plan from a previous compilation is operable. The SQL compiler then recompiles only the statements that fail the similarity check; the other SQL statements retain their existing plans.

The COMPILE option has three forms:

- COMPILE PROGRAM directs the SQL compiler to explicitly compile all SQL statements in the program. COMPILE PROGRAM is the default.

  If you include the STORE SIMILARITY INFO clause, the SQL compiler stores similarity information for each SQL statement in the program file.

- COMPILE INVALID PLANS directs the SQL compiler to explicitly compile these SQL statements:
    - Statements that refer to changed DEFINEs.
    - Statements with plans that fail the redefinition timestamp check.
    - Statements with altered execution plans. An altered execution plan is an invalid but operable plan that the SQL compiler has updated without recompiling. For more information, see Altered Execution Plans.
    - Uncompiled SQL statements with empty sections. The SQL compiler generates an empty section if an SQL statement refers to a nonexistent DEFINE or SQL object.

  Other SQL statements retain their existing execution plans.

  The COMPILE INVALID PLANS option stores similarity information in the program file and updates the program’s name map and usages in the USAGES tables.

  If the program has not been previously compiled or if the program does not contain similarity information, the COMPILE INVALID PLANS option directs the SQL compiler to compile all SQL statements in the program.

- COMPILE INOPERABLE PLANS directs the SQL compiler to explicitly compile these SQL statements:
    - Statements with inoperable plans (plans that fail the similarity check).
    - Uncompiled statements with empty sections. The SQL compiler generates an empty section if an SQL statement refers to a nonexistent DEFINE or SQL object. (The SQL compiler also generates empty sections for CONTROL directives and DDL statements.)

  Other SQL statements retain their existing execution plans.

  The COMPILE INOPERABLE PLANS option stores similarity information in the program file and updates the program’s name map and usages in the USAGES tables.

  If the program has not been previously compiled or if the program does not contain similarity information, the COMPILE INOPERABLE PLANS option directs the SQL compiler to compile all SQL statements in the program.

Altered Execution Plans

If you recompile a program by using the COMPILE INOPERABLE PLANS option, the SQL compiler performs the similarity check if an SQL object refers to a changed DEFINE or the timestamp check fails for a referenced object in the execution plan.
If the similarity check passes, the SQL compiler alters the execution plan with this new information:

- Physical name
- Redefinition timestamp. The new timestamp prevents future similarity checks for the plan until the SQL object changes again.
- Partition node array. SQL uses the partition node array to determine alternate paths when partitions are unavailable to a plan.

SQL considers an altered execution plan to be invalid but operable and recompiles the plan as follows:

- During explicit compilation, the SQL compiler recompiles an altered plan if you specify the COMPILE PROGRAM or COMPILE INVALID PLANS option.
- At run time, the SQL executor automatically recompiles an altered plan if you specified the CHECK INVALID PROGRAM or CHECK INVALID PLANS option during the previous explicit SQL compilation.

CURRENTDEFINES and STOREDDEFINES Options

If you recompile a program using the CURRENTDEFINES option (which is the default) and a statement refers to a DEFINE that does not exist, the SQL compiler recompiles the statement and generates an empty section, regardless of whether the previous execution plan was valid.

If you recompile a program by using the STOREDDEFINES option, the SQL compiler resolves DEFINE names using the values stored in the program’s name map during the previous explicit compilation. In this case, the current DEFINE values have no effect on the COMPILE INVALID PLANS and COMPILE INOPERABLE PLANS options, and the STOREDDEFINES option does not change the name map in the program file.

Example: Explicitly Recompiling Only Inoperable Plans

If you must explicitly recompile a program, but you want to minimize downtime for the program, use the COMPILE INOPERABLE PLANS option and the similarity check to recompile only the statements with inoperable plans that fail the similarity check.

Follow these steps to implement this solution:

1. Enable the similarity check for each table or protection view specified in the SQL statements as follows:
    - For existing tables, use the ALTER TABLE or ALTER VIEW statement with the SIMILARITY CHECK ENABLE clause.
    - If you are creating a new table or protection view, use the CREATE TABLE or CREATE VIEW statement with the SIMILARITY CHECK ENABLE clause.
2. Explicitly SQL compile the program with the COMPILE INOPERABLE PLANS option. The SQL compiler compiles only the SQL statements with invalid plans that fail the similarity check (and any uncompiled statements).

If the program has not been previously compiled or does not contain similarity information, the COMPILE INOPERABLE PLANS option directs the SQL compiler to compile all SQL statements in the program.

New Indexes

If you add any new indexes, you might decide to explicitly SQL compile the program with the COMPILE INVALID PLANS option. The SQL compiler then recompiles the SQL statements that refer to the tables affected by the new indexes. Consequently, the compiler might generate new and more efficient execution plans that use the new indexes.

New SQL Compiler Version

If you have installed a new version of the SQL compiler since the last explicit compilation, you might decide to explicitly SQL compile the program with the COMPILE PROGRAM STORE SIMILARITY INFO option. The SQL compiler recompiles all SQL statements and stores similarity information in the program file. Also, the new compiler might generate more efficient execution plans.

Enabling the Similarity Check for Tables and Protection Views

You must explicitly enable the similarity check for a table or protection view (including any underlying tables for the view) to use these options. (SQL implicitly enables the similarity check for other SQL objects.)

- CHECK options: To use the CHECK INOPERABLE PLANS option, the similarity check must be enabled for any tables or protection views referenced at run time.
- COMPILE options: To use the COMPILE INOPERABLE PLANS option, the similarity check must be enabled for any tables or protection views referenced during explicit SQL compilation.

To enable or disable the similarity check for a table or protection view, specify the SIMILARITY CHECK clause in the CREATE or ALTER TABLE or VIEW statements. For the complete syntax of these statements, see the *NonStop SQL/MP Reference Manual*.

Invalidation of Programs

If you use the ALTER TABLE or ALTER VIEW statement to change the similarity check attribute, the SQL catalog manager invalidates any programs, as identified in the USAGES table, that refer to the table or protection view. If the ALTER TABLE or ALTER VIEW statement sets the similarity check attribute to its current value, programs are not invalidated.

Underlying Tables

If you enable the similarity check for a protection view, the operation does not enable the check for any underlying tables. You must explicitly enable the similarity check for the underlying table. If you enable the similarity check for an underlying table, the operation does not enable the check for a protection view defined on the table.

Collations

You do not have to enable the similarity check for a collation, because collations always have the similarity check enabled. Collations are similar only if they are equal. SQL uses the CPRL_COMPAREOBJECTS_ procedure to compare the collations. Consequently, two tables that contain character columns associated with collations are similar only if the collations are equal.

Similarity Rules for Tables

There are two separate comparison situations that apply to similarity checking:

- Static compilation, with previously compiled access to a table and a current compilation that accesses a table
- Execution time, with previously compiled access to a table and current access to a table

For two tables to be similar, the characteristics and attributes of the tables must be the same except for a specific set of allowable differences, such as:

- Names of the tables
- Contents of the tables (that is, the data in the table)
- Partitioning attributes (number of partitions and partitioning key ranges)
- Number of indexes. RUN-TIME-TABLE must have all indexes used by COMPILE-TIME-TABLE in the execution plan. RUN-TIME-TABLE can also have additional indexes that COMPILE-TIME-TABLE does not have. COMPILE-TIME-TABLE can have indexes that RUN-TIME-TABLE does not have but only if the execution plan does not use the additional indexes.
- Key tags (or values) for indexes
- Creation timestamp and redefinition timestamp
- AUDIT attribute. If, however, a statement performs a DELETE or UPDATE set operation on a nonaudited table with a SYNCDEPTH of 1, the SQL executor returns an error and forces the automatic recompilation of the statement (if NORECOMPILE is not specified).

- Any of these file attributes:

ALLOCATE	LOCKLENGTH	SECURE
AUDITCOMPRESS	MAXEXTENTS	SERIALWRITES
BUFFERED	NOPURGEUNTIL	TABLECODE
CLEARONPURGE	OWNER	VERIFIEDWRITES
EXTENT (primary and secondary)		

- Statistics on the tables
- Column headings
- Comments on columns, constraints, collations, indexes, or tables
- Catalog where the table is registered
- Help text
- Number of columns. RUN-TIME-TABLE can have more columns than COMPILE-TIME-TABLE, but the common columns of both tables must have identical attributes. However, if a statement uses a SELECT list containing an asterisk (*), RUN-TIME-TABLE must have the same number of columns as COMPILE-TIME-TABLE. For more information, see the following subsections.

Note. The similarity check does not apply to parallel execution plans. Tables are not considered similar if they are specified in a query that uses a parallel execution plan.

For more information about similarity, see the SQL/MP programming manual for your host language.

Similarity Rules for Protection Views

The similarity check does not support shorthand views; similarity rules for protection views are:

- A protection view is never similar to a table or any other object.
- To pass the similarity check, two protection views must follow these criteria:
  - Have similar underlying base tables
  - Project the same columns from the base tables
  - Have the same column names
  - Have the same selection expression, which is determined by a binary comparison of the generated objects for the two selection expressions

Planning for TS/MP Requirements

Transaction processing applications can access both SQL and Enscribe databases. To access an SQL database, the servers must use embedded SQL statements coded in host language programs such as COBOL, C, or Pascal. To access an Enscribe database, servers use languages, such as COBOL, FORTRAN, C, or Pascal, or TAL statements, with calls to Enscribe I/O procedures. A single server can access both an SQL database and an Enscribe database with the appropriate database statement for each type of table or file access.

To access either an SQL or Enscribe database from a TMF environment, programs can use logical names with the actual file names specified at compile or run time. To use logical names for an SQL database, use SET SERVER DEFINE commands coded in the PATHMON configuration file. To use logical names for Enscribe files, use SET SERVER DEFINE or SET SERVER ASSIGN commands in the PATHMON configuration file. Servers can use hard-coded names (table or view names for SQL or file names for Enscribe) to access either database.

When you are running HP NonStop TS/MP, make sure that the security of the SQL objects allows the user ID running TS/MP to access the same tables and views that the servers use.

If you want to log SQL compilations and automatic recompilations for your application environment, you must add the =_SQL_CMP_EVENT DEFINE mapped to a disk file. The disk file must be accessible by the user ID under which the applications are running. Recompilations are automatically logged to $0 unless you either turn logging off or designate a different device with a DEFINE. For more information on logging, see Monitoring Compilations on page 10-5.

Planning for Pathmaker Requirements

Three column attributes are particularly useful for applications generated by the Pathmaker application development tool. These are the HEADING, HELP TEXT, and UPSHIFT attributes. If these attributes are assigned when the columns are defined, the Pathmaker tool automatically incorporates their use in the DB requesters.

- The HEADING attribute assigns an alternate heading for the column. If no heading is assigned, however, the Pathmaker tool uses the column name as the default heading.
- The HELP TEXT attribute associates help text, which is stored in the COMMENTS table, with the column. When building an application, the Pathmaker tool retrieves the help text from COMMENTS and inserts the text into its HELP facility.
- The UPSHIFT attribute signifies that the data will automatically be upshifted before storage in the column. UPSHIFT is supported for single-byte character data types only.

For additional information about how the Pathmaker tool uses these column attributes, see the *Pathmaker Reference Manual* and the *Pathmaker Programming Guide*.

Using DEFINEs

A DEFINE is a named set of attributes and associated values stored in the process file segment (PFS) of a running process.

By using logical names, you can run SQLCI commands and host programs using different sets of files, depending on the DEFINEs in effect at the time. Application code that uses logical names instead of hard-coded file names can be compiled or executed to use different physical databases.

You can use a DEFINE name to specify a subvolume, catalog, table, view, collation, or index in SQL/MP statements and commands and in host application programs. Typically, you use a DEFINE to establish a one-to-one mapping between a logical name and a physical name. When combined with execution-time name resolution, you can use DEFINEs to run a program against a different database or dynamically select a database at run time.

There are several ways to use DEFINEs with SQL/MP at compile time, from SQLCI, and from within programs.

Entering DEFINE Commands

To use DEFINEs, preset the DEFINE names and attribute values by entering DEFINE commands at the command interpreter or SQLCI prompt or by using Guardian DEFINE procedures in a host language program. When you run an SQL statement or SQLCI command that includes a DEFINE name, the system substitutes the name of the actual object for the logical name.

The SQL software uses DEFINEs to override default processing for SQL components. For information about these DEFINEs, see the *SQL/MP Reference Manual*.

DEFINE Rules

These rules apply when using DEFINEs in applications and in SQLCI:

- A DEFINE name can contain from 2 to 24 characters; the first character must be an equal sign (=), and the second must be a letter (for user-defined names). After the first two characters, the name can contain any combination of alphanumeric characters, hyphens (-), underscores (_), and circumflexes (^).

Note. A special class of DEFINE names begins with an equal sign and an underscore (=_). These names are reserved for HP use only. Do not attempt to create DEFINE names that begin with these two characters unless specifically directed to do so in NonStop system documentation.

Some examples of DEFINE names follow:

```
=EMPLOYEE
=DEPT_MGR_NAMES
=AR_CATALOG
=MGR_PROTECTION_VIEW
```

- The DEFMODE option must be set to ON to enable adding DEFINEs. The DEFMODE setting remains in effect for the duration of the command interpreter or SQLCI session in which the setting is established.

  These are the DEFMODE settings and the commands to set them:

SET DEFMODE ON	Enables DEFINEs; a new process inherits the DEFINE set from the initiating process. This setting is the default value.
SET DEFMODE OFF	Disables DEFINEs; a new process inherits only the =_DEFAULTS DEFINE from the initiating process.

- SQL/MP statements and commands recognize DEFINEs but do not recognize file names set by the command interpreter ASSIGN command.

General Guidelines

These general guidelines apply to using DEFINEs:

- Using DEFINEs that identify the wrong objects causes errors or ambiguity. In general, the user cannot determine the effects of using a DEFINE at run time; therefore, a query or program could access the wrong SQL object if security is not properly set. Be sure that the DEFINEs in effect identify the objects you want to use.
- If the DEFMODE option is set to ON when you start an SQLCI session, SQLCI inherits all the DEFINEs already set for the command interpreter process.
- Putting DEFINE commands in an OBEY command file, by using a text editor, is a simple way to ensure the consistency of the environment. To set DEFINEs, place the SET DEFMODE ON command first, and then include an ADD DEFINE or ALTER DEFINE command for each DEFINE. If the DEFINE set is shared by multiple users, each user must run the OBEY command file at the command interpreter prompt.
- After you create a DEFINE, it stays in effect until you change it with the ALTER DEFINE command, delete it with the DELETE DEFINE command, disable the use of DEFINEs in the current SQLCI session with the SET DEFMODE OFF command, or end the session or the process. If you create DEFINEs in an SQLCI session, they are released when the session ends. If you create the DEFINEs in a command interpreter session, they remain in effect until you delete or alter them or until you log off.

- In SQLCI, the DELETE DEFINE, ALTER DEFINE, and ADD DEFINE commands apply only to the DEFINE set for the SQLCI session. For example, in an SQLCI session, if you alter a DEFINE inherited from the command interpreter process, the DEFINE is altered only for the SQLCI session. When you return to the command interpreter prompt after ending the SQLCI session, the original DEFINE is still in effect.

Using DEFINEs During Compilation

Use DEFINEs to establish various attributes and other values during compilation. These compiler options control the DEFINE set in effect during the compilation:

- CURRENTDEFINES— selects the set of DEFINEs in effect for logical name mapping at the time of the last explicit or automatic compilation. This option is the default.
- STOREDDEFINES— selects the DEFINE name mapping stored with the program from the last explicit SQL compilation.

  The SQL compiler stores the DEFINEs current at explicit compile time in the program code.

  Your choice of compiler option for DEFINEs depends on the situation. If you are explicitly compiling a program because a table has a new index, you should use the STOREDDEFINES option. This option ensures that all the original DEFINEs are used, thereby eliminating the possibility of setting incorrect DEFINEs. You should not use STOREDDEFINES when compiling a program for the first time, because the program has no stored DEFINEs.

  If you are explicitly compiling a program because a table has been moved, you should use the CURRENTDEFINES to enable the current set of DEFINEs to identify the new location of the table.

Using DEFINE Names With Programs

Use DEFINEs to establish various attributes and other values for your programs to use. These guidelines apply to using DEFINEs:

- To refer to DEFINEs within your programs, you must create the DEFINEs during your command interpreter session with the DEFMODE option set to ON. The DEFINE set is inherited by the compilers and programs that are subsequently started. DEFINEs that are referred to in a program are evaluated, and physical file names are resolved first, at compile-time.

  If your application executes in a Pathway transaction processing environment, however, you specify DEFINE names for a server class by using the SET SERVER DEFINE command, as described under Using DEFINEs With PATHMON on page 10-37.

- After a program that uses DEFINE names is SQL compiled and registered as a valid program in a catalog, the program is valid only for the table and views identified by the DEFINEs at compile time. If a different set of DEFINEs is used at run time, the program is automatically recompiled with the new DEFINEs if automatic recompilation is enabled.

  If you want the program to be valid for a different table or view, you must SQL compile the program with the new DEFINEs to revalidate the program with the objects identified by the new DEFINEs.

- You must know which DEFINEs are required in the run time environment: only those DEFINEs that refer to a catalog or SQL object are required at run time. These DEFINEs are optional at SQL compile time. Other DEFINEs used in programming, such as =COPYLIB (referring to COPY libraries of the source code), are already resolved by the precompilers and are not part of the run time DEFINE set.

- The EXPLAIN DEFINES option of the SQL compiler lists the DEFINE set used to compile the program. You can access this listing in an OBEY command-file format to use before executing the programs.

- Using DEFINEs simplifies mobility issues for application programs. Using DEFINEs, however, can create problems if the wrong DEFINE is active at run time. If the incorrect DEFINE is in effect and identifies a table or view that exists, the program could write to the wrong table. This error could happen most easily on a system shared by groups using similar databases when security is not planned carefully.

Examples

To demonstrate the use of DEFINEs with programs, several examples follow.

The first example uses a DEFINE in an INVOKE statement of a COBOL program. The logical name =PARTS identifies a table. The DEFINE for =PARTS must be in effect only during the preprocessing step of the COBOL compilation.

```
EXEC SQL
 INVOKE =PARTS AS PARTS-REC LEVEL (01,04)
END-EXEC.
```

This example uses the logical name =PARTS in an INSERT statement. The DEFINE for =PARTS must be in effect to identify the table when this statement is SQL compiled and executed.

```
EXEC SQL
 INSERT INTO =PARTS
   VALUES (:PARTNUM  OF PARTS,
          :PARTDESC OF PARTS,
          :PRICE    OF PARTS,
          :QTY-AVAILABLE OF PARTS )
END-EXEC.
```

This example uses a logical name to identify a COBOL library. The logical name is resolved only at preprocess time when the associated information is copied into the program.

```
EXEC SQL BEGIN DECLARE SECTION       END-EXEC.
   EXEC SQL
        SOURCE  =COPYLIB(DEPT, JOB)  END-EXEC.
EXEC SQL END DECLARE SECTION         END-EXEC.
```

Using DEFINEs From SQLCI

These examples illustrate the use of DEFINEs with SQLCI.

These commands first set the DEFMODE option to ON and then create DEFINEs in an SQLCI session:

```
>> SET DEFMODE ON;
>> ADD DEFINE =AR_CATALOG, CLASS CATALOG,
+>            SUBVOL \SYS1.$VOL.ARCAT;
>> ADD DEFINE =EMPLOYEE, CLASS MAP,
+>            FILE \SYS1.$VOL1.PERSNL.EMPS;
>> ADD DEFINE =MGR_PROTECTION_VIEW, CLASS MAP,
+>            FILE \SYS1.$VOL1.PERSNL.EMPSP1;
>> ADD DEFINE =PR_CATALOG, LIKE =AR_CATALOG,
+>            SUBVOL \SYS1.$VOL.PRCAT;
```

These commands alter DEFINEs in an SQLCI session:

```
>> ALTER DEFINE =AR_CATALOG, SUBVOL \SYS1.$NEWVOL.ARCAT;
>> ALTER DEFINE =EMPLOYEE, FILE \SYS1.$NEWVOL.PERSNL.EMPS;
```

These commands delete DEFINEs in an SQLCI session:

```
>> DELETE DEFINE =AR_CATALOG;
>> DELETE DEFINE (=EMPLOYEE,=MGR_PROTECTION_VIEW);
```

These commands alter the =_DEFAULTS DEFINE in an SQLCI session:

```
>> VOLUME \SYS1.$VOL1.PERSNL;
>> CATALOG \SYS1.$VOL1.PERSNL;
```

You can also use an ALTER DEFINE command to alter the =_DEFAULTS DEFINE as shown:

```
>> ALTER DEFINE  =_DEFAULTS, CATALOG \SYS1.$VOL1.SALES;
```

This example uses DEFINEs in the CREATE TABLE statement to identify the table and catalog. Suppose that the DEFINEs were previously added as shown in the preceding examples:

```
>> CREATE TABLE =EMPLOYEE
+>       (EMP_NUM      PIC 9(6)   DEFAULT SYSTEM   NOT NULL,
+>        EMP_NAME     PIC X(30)  NO DEFAULT       NOT NULL,
+>        SS_NUMBER    PIC X(11)  NO DEFAULT       NOT NULL,
+>        ADDRESS      PIC X(30)  DEFAULT SYSTEM   NOT NULL,
+>        CITY         PIC X(30)  DEFAULT SYSTEM   NOT NULL,
+>        ST           PIC X(2)   DEFAULT SYSTEM   NOT NULL,
+>        ZIP_CODE     PIC X(5)   DEFAULT SYSTEM   NOT NULL,
+>        PRIMARY KEY EMP_NUM)
+> CATALOG =PR_CATALOG;
--- SQL operation complete.
```

Using DEFINEs to Switch Databases

By using the similarity check and DEFINEs, you can run programs against different databases or dynamically select a database without auto-recompilation.

Running a Program Against Different Databases

This scenario describes a situation where you explicitly SQL compile a program using a specific database. Several users, each with a different but similar database, want to run the program. Each user wants to specify a set of DEFINEs that point to the respective new database.

To allow access to different databases, specify DEFINEs for all tables and protection views and use the similarity check to avoid automatic recompilation.

Follow these steps to implement this solution:

1. Specify DEFINEs for all tables and protection views used in the SQL statements. These DEFINEs should point to tables and protection views in the first database.
2. Explicitly SQL compile the program with the CHECK INOPERABLE PLANS option to enable the similarity check for the program.

Each user should then perform these steps:

1. Enable the similarity check for each table or protection view specified in the SQL statements as follows:
   - For existing tables, use the ALTER TABLE or ALTER VIEW statement with the SIMILARITY CHECK ENABLE clause.
   - If you are creating a new table or protection view, use the CREATE TABLE or CREATE VIEW statement with the SIMILARITY CHECK ENABLE clause.
2. Run the program with DEFINEs that point to the new database. The SQL executor uses the similarity check to compare the original tables with the new tables. If the similarity check passes for an SQL statement, the SQL executor executes the statement without recompiling it. (The usage information is available only for the original tables specified during the explicit SQL compilation.)

Dynamically Selecting Different Databases

This scenario describes a situation where you have several similar SQL/MP databases and you want a program to dynamically determine which database to access. For example, your program might select the database depending on the type of transaction. You do not want to use dynamic SQL statements because they require extra programming time to write and can degrade your node's performance during execution. You could combine all the databases into a single database, but the management of a large database would be complicated.

To provide dynamic access, specify DEFINEs for all table names and then use execution-time name resolution and the similarity check.

Follow these steps to implement this solution:

1. Modify the program as follows:
    - Specify the CONTROL QUERY BIND NAMES AT EXECUTION directive in the source file to enable execution-time name resolution for all DML statements.

      You might need to specify more than one directive depending on the structure of your program and the scoping rules for the host language you are using. For more information, see The *SQL/MP Reference Manual*.
    - Use DEFINEs in all SQL DML statements.
    - Add source code that alters the DEFINEs used in each SQL statement to point to the appropriate database when the SQL statement is executed.
2. Enable the similarity check for each table or protection view specified in the SQL statements as follows:
    - For existing tables, use the ALTER TABLE or ALTER VIEW statement with the SIMILARITY CHECK ENABLE clause.
    - If you are creating a new table or protection view, use the CREATE TABLE or CREATE VIEW statement with the SIMILARITY CHECK ENABLE clause.
3. Explicitly SQL compile the program with the CHECK INOPERABLE PLANS option to enable the similarity check for the program.
4. Run the program. The program uses the different DEFINE values to determine the database to access. The SQL executor resolves the DEFINE names at statement execution time and executes the similarity check to prevent automatic recompilation.

Using DEFINEs With PATHMON

If your application executes in a Pathway transaction processing environment, you specify DEFINE names for a server class by using the SET SERVER DEFINE command. SQL tables and views are referred to by using DEFINE names. When you specify DEFINEs within the server configuration, you are associating DEFINE names used by that server class with actual physical files.

You should indicate the maximum number of DEFINEs that will exist across all server class definitions by entering a SET PATHWAY MAXDEFINES command. The Pathway transaction processing environment must be cold started to set this attribute. Be sure to allow an ample number of DEFINEs for your environment. If you do not specify a value for MAXDEFINES, the default value is 0.

Suppose that you expect a total of 53 DEFINEs for your application. You indicate the maximum number of DEFINEs by specifying this command in the PATHMON configuration file:

```
= SET PATHWAY MAXDEFINES 53
```

Suppose that a server class, SRV-SDB102, contains SQL statements that refer to these DEFINE names: =EMP, =DEPT, and =JOB. To associate these DEFINE names with a physical object in your database, include these commands in your server class configuration:

```
= SET SERVER DEFINE =EMP  , CLASS MAP, &
=   FILE \SYS1.$VOL1.TEST1.EMPLOYEE
= SET SERVER DEFINE =DEPT , CLASS MAP, &
=   FILE \SYS1.$VOL1.TEST1.DEPT
= SET SERVER DEFINE =JOB  , CLASS MAP, &
=   FILE \SYS1.$VOL1.TEST1.JOB
    .
    .
= ADD SERVER SRV-SDB102
```

When a server process in the server class SRV-SDB102 starts, the specified DEFINE definitions are included as part of the process environment. The system uses the information in the DEFINE definitions when the server refers to a DEFINE by name. For MAP DEFINEs or CATALOG DEFINEs, this approach results in the substitution of the associated object or catalog name for the DEFINE name in the SQL statement at run time.

Altering PATHMON DEFINEs

You can change DEFINEs without stopping the PATHMON environment. You must stop the server class, however, to alter the DEFINEs it uses. To stop the server class, you must freeze it.

Suppose that during development you must move the referenced objects to a new volume. You can use the ALTER SERVER commands to replace the existing DEFINEs with new DEFINEs, as shown:

```
= FREEZE SERVER SRV-SDB102
= STOP SERVER SRV-SDB102
= ALTER SRV-SDB102, DEFINE =EMP , CLASS MAP, &
=   FILE \SYS1.$VOL2.TEST2.EMPLOYEE
= ALTER SRV-SDB102, DEFINE =DEPT, CLASS MAP, &
=   FILE \SYS1.$VOL2.TEST2.DEPT
= ALTER SRV-SDB102, DEFINE =JOB , CLASS MAP, &
=   FILE \SYS1.$VOL2.TEST2.JOB
= THAW SERVER SRV-SDB102
= START SERVER SRV-SDB102
```

You can delete the existing DEFINEs for a server with these commands:

```
= FREEZE SERVER SRV-SDB102
= STOP SERVER SRV-SDB102
= ALTER SRV-SDB102, DELETE DEFINE =EMP
= ALTER SRV-SDB102, DELETE DEFINE =DEPT
= ALTER SRV-SDB102, DELETE DEFINE =JOB
= THAW SERVER SRV-SDB102
= START SERVER SRV-SDB102
```

After you alter DEFINEs for a server, automatic recompilation occurs every time the server is started. For this reason, you might want to explicitly SQL compile the server after altering DEFINEs.

You do not need to cold start the PATHMON environment when you alter the DEFINE set for the server classes. To change the MAXDEFINES attribute, however, you must cold start the Pathway transaction processing environment.

The PATHMON process manages the active set of DEFINEs while TS/MP is running. DEFINEs in this environment are completely separate from DEFINEs associated with the command interpreter process or other processes. You can alter DEFINEs outside of the transaction processing environment without affecting the active set for your TS/MP applications.

Manipulating Program Files

SQL program files can be manipulated just like other program files:

- SQL programs stored in Guardian files can be objects of the FUP commands RENAME, PURGE, and SECURE; the TACL commands RENAME and PURGE; and the system procedures FILE_RENAME_ , FILE_PURGE_ , and SETMODE.
- SQL programs stored in OSS files are manipulated by using OSS utilities such as `rm` and `mv`. For more information about OSS utilities, see the *Open System Services Shell and Utilities Reference Manual*.

SQL program files, unlike other SQL objects, can reside on nonaudited volumes.

SQL DDL statements that require all other dependent objects to be available, such as DROP TABLE, can complete when a dependent program is not available if the program’s catalog is available. If the program file is available, these DDL operations invalidate the programs by registering the invalidation in the catalogs and program file label. If the program file is not available, the invalidation is registered in the program's catalog and detected at run time.

Moving Programs

Moving a program is similar to moving other SQL objects, but somewhat easier because programs have no dependencies. You can use either FUP DUP or SQLCI DUP or the BACKUP and RESTORE utilities to move SQL programs stored in Guardian SQL files. To move SQL programs stored in OSS files, use the appropriate OSS utility. (For more information about OSS utilities, see the *Open System Services Shell and Utilities Reference Manual*.)

Normally, when you move a program the new program is not registered in a target catalog and authority to write to the catalog is not required for the move operation. The new file is created with the SQL SENSITIVE flag turned off so that the new program file is no longer a valid SQL program. If you specified a similarity check during the original compilation of your program, you do not need to recompile your program. Otherwise, you must explicitly SQL compile the program files after a move to validate the programs and register them in a catalog.

Programs are easily moved between databases in these situations:

- Logical names have been used in the programs. Logical names in a program make program code independent of the location of the database so that the program can be compiled with a new set of DEFINEs and validated to a new database.
- The program accesses similar objects and can run without recompilation.

These paragraphs describe the use of compiler options that affect recompilation when moving programs, followed by information about using RESTORE and SQLCI DUP to move SQL programs stored in Guardian files. For a comparison of moving SQL objects with the SQLCI DUP utility and with the BACKUP and RESTORE utilities, see Moving Database Objects on page 9-14.

△ **Caution.** If an SQL object has the UNRECLAIMED FREESPACE (F) or INCOMPLETE SQLDDL OPERATION (D) attribute set, do not attempt to back up, move, or duplicate the object until the attribute is reset. For more information, see UNRECLAIMED FREESPACE (F) and INCOMPLETE SQLDDL OPERATION (D) Flags on page 7-24.

Moving Programs Without Recompilation

These SQL compiler options can be useful in the management of SQL programs:

- The REGISTERONLY option directs the SQL compiler to register a previously SQL compiled program in a specific catalog without recompiling any SQL statements in the program. You can use this option to install a program in a catalog after you have SQL compiled and moved the program. Although the REGISTERONLY option requires you to run the compiler, this option is more efficient than explicitly recompiling the entire program.

  If the REGISTERONLY ON option is used, the SQL compiler summary listing specifies that the SQL statements were not compiled and the plans are unchanged.

- The NOREGISTER option directs the SQL compiler to compile a program without registering the program in a catalog. You can then move the program by using a FUP or SQLCI DUP command or the BACKUP and RESTORE programs. After the move, you can run the program without recompiling or registering it in a catalog.

  A program compiled with the NOREGISTER ON option can never be registered in a catalog. If you try to register a program compiled with the NOREGISTER ON option by using the REGISTERONLY ON option, the operation fails with an SQL error. If a program was compiled with the NOREGISTER ON option and you need to register the program in a catalog, you must explicitly recompile it with the REGISTERONLY and NOREGISTER options set to OFF (or without these options altogether, which is the default).

  If the NOREGISTER ON option is used, the SQL compiler summary listing specifies that the program is not registered in a catalog.

Note. The REGISTERONLY and NOREGISTER options are mutually exclusive options.

The REGISTERONLY and NOREGISTER options operate independently of similarity checking, but the use of similarity checking with these options makes a move operation more efficient by minimizing recompilations.

If you install a program with the REGISTERONLY option and the program was not previously compiled with the CHECK INOPERABLE PLANS option, the SQL executor forces the automatic recompilation of static SQL statements in the program unless the program accesses the same tables at run time that it accessed during explicit SQL compilation. This restriction does not apply to dynamic SQL statements.

Restrictions for the NOREGISTER Option

These static SQL statements, when compiled with the NOREGISTER ON option, must use execution-time name resolution, or the SQL compilation fails with SQL error 2109:

- DML statements: SELECT, INSERT, UPDATE, DELETE, and DECLARE CURSOR
- LOCK and UNLOCK statements
- GET VERSION and GET CATALOG statements

To prevent automatic SQL recompilation of these statements at SQL load time, specify the CHECK INOPERABLE PLANS option during explicit compilation (if the SQL names used in the statements are changed after explicit SQL compilation).

Example: Installing a Program at a New Location Without Recompilation

The next example describes a scenario where you might develop a program on a development system and then move the program to a production system. You do not want new plans to be generated for the production system. You would like to avoid a required compilation on the production system. The recompilation causes downtime for the program and degrades the performance for the system.

To avoid recompilation, enable the similarity check for the program and any referenced tables or protection views. After compiling the program on the development system, move it to the production system and then install it by using the REGISTERONLY option. Follow these steps:

1. On the development system, explicitly SQL compile the program using the CHECK INOPERABLE PLANS option. Specify DEFINEs that point to objects on the development system. You are not required to enable the similarity check for the tables or protection views on the development system.
2. Move the program to the production system by using the FUP or SQLCI DUP command or the BACKUP and RESTORE program.
3. On the production system, enable the similarity check for each table or protection view specified in the SQL statements as follows:
    - For existing tables, use the ALTER TABLE or ALTER VIEW statement with the SIMILARITY CHECK ENABLE clause.
    - If you are creating a new table or protection view, use the CREATE TABLE or CREATE VIEW statement with the SIMILARITY CHECK ENABLE clause.
4. On the production system, run SQLCOMP with the REGISTERONLY ON option to register the program in an SQL catalog. Specify a catalog name, if you wish, or use the default catalog. This operation is much faster than explicitly compiling the entire program, because it does not generate new execution plans.

The REGISTERONLY ON option does not generate usages in the USAGES table. If you require usages on the production system, you must explicitly recompile the program. If you do recompile the program, specify the COMPILE INOPERABLE PLANS option to improve performance.

5. Run the program with DEFINEs that point to the objects on the production system. The SQL executor uses the similarity check to compare the production tables with the development tables. If the similarity check passes for an SQL statement, the SQL executor executes the statement without recompiling it.

Example: Moving a Program Without Registering It on the New System

The next example describes a scenario where you want to compile a program on one system and then move it to a different system before you run it. You do not require that the program be registered in the catalog on the second system. To eliminate recompilation on the new system, use execution-time name resolution for all DML statements and compile the program with the NOREGISTER ON option before you move it to the new system.

Follow these steps:

1. Specify the CONTROL QUERY BIND NAMES AT EXECUTION directive in the program’s source file to enable execution-time name resolution for all DML statements.

   You might need to specify this directive more than once, depending on the structure of your program and the scoping rules for the host language you are using. For more information, see Deferring Name Resolution on page 10-13.

2. Explicitly SQL compile the program with the NOREGISTER ON option.

3. Move the program to the other systems by using the FUP or SQLCI DUP command or the BACKUP and RESTORE programs.

4. Run the program. The NOREGISTER option enables the program to run although the program is not recompiled on the new system or registered in the catalog.

Using BACKUP and RESTORE

To use the BACKUP and RESTORE programs to move an SQL program file from one node to another node:

1. On the first node, back up the program.

2. On the second node, restore the program using the SQLCOMPILE ON and REGISTERONLY ON options. The program is restored and registered on the second node without being recompiled.

RESTORE Program and the SQLCOMPILE Option

If you restore a program using the SQLCOMPILE option, the RESTORE program invokes the recompilation of the program using the SQLCOMP CHECK option specified during the last explicit SQL compilation.

The SQLCOMPILE ON option of RESTORE can restore a program and automatically recompile the program. The program is explicitly recompiled with the DEFINEs stored in the program description, the same effect as specifying STOREDDEFINES in the SQLCOMP command. You would use the SQLCOMPILE ON option in recovery operations but not for moving programs from one database to another.

This example uses RESTORE to restore a single SQL program stored in a Guardian file. The example then uses the SQLCOMPILE ON option to request automatic recompilation of the program:

```
RESTORE $TAPE, $VOL1.PERSNL.EMPPROG,
        MAP NAMES ($VOL1.PERSNL.EMPPROG to $VOL1.ADMIN.*),
        SQLCOMPILE ON, LISTALL
```

RESTORE Program and the REGISTERONLY ON Option

The REGISTERONLY ON option adds a few considerations to the use of RESTORE to restore programs. Consider an SQL program explicitly compiled with the REGISTERONLY ON option (the initial compilation) and then backed up by using the BACKUP program. If you restore this program by using the SQLCOMPILE ON option, the RESTORE program invokes the SQL recompilation of the program using the SQLCOMP options specified during the explicit SQL compilation (that is, the explicit SQL compilation immediately before the compilation using the REGISTERONLY ON option).

When using the REGISTERONLY option with the RESTORE program, the SQL program is not recompiled, but is restored and registered in the catalog.

RESTORE Program and the NOREGISTER ON Option

The NOREGISTER ON option causes an SQL program file to appear as an Enscribe file to the RESTORE program. Therefore, if you restore a program file using the SQLCOMPILE option, the RESTORE program does not invoke the SQL compiler for an SQL program file compiled with the NOREGISTER ON option.

Using SQLCI DUP

This example shows using the SQLCI DUP command to move a single SQL program stored in a Guardian file. After the move, the user requests explicit compilation with registration in catalog $VOL1.ADMIN:

```
>> DUP $VOL1.PERSNL.EMPPROG, $VOL1.ADMIN.*,
+>      SAVEALL;
>> EXIT;
21> SQLCOMP /IN $VOL1.ADMIN.EMPPROG, OUT $S.#HOLD/
             CATALOG $VOL1.ADMIN
```

If you do not need to recompile your programs, omit the second step.

If you have stored your programs in a separate subvolume, you can use the wild-card character as the file set list in the DUP command to efficiently duplicate groups of Guardian programs.

This example shows how to use the wild-card character * (asterisk) to move sets of programs. The SAVEALL option creates the target file with the same security, owner, and timestamps as the corresponding source file.

```
>> DUP $VOL1.PERSNL.*, $VOL1.ADMIN.*,
+>     SAVEALL;
```

The SQLCI DUP command supports the qualified file-set list so that only program files are identified by the list. This example uses a qualified file-set list to identify and move a set of programs from a specified catalog:

```
>> DUP $VOL1.PERSNL.* WHERE SQLPROGRAM,
+>     MAP NAMES ($VOL1.PERSNL.* TO $VOL1.ADMIN.*), SAVEALL;
```

You might store programs in many subvolumes throughout the system. In this case, you can specify a qualified file-set list to move all the programs from one or many catalogs; however, you must also include a detailed MAP NAMES and CATALOG specification to handle all the cases.

If the mapping strategy is complex, you can use other methods for moving programs. These two examples show two methods of using the DUP command to move these programs:

```
$OLD1.PROGS.PROGA1, described in catalog $OLD1.ACCTG
$OLD2.PROGS.PROGA2, described in catalog $OLD1.ACCTG
$OLD3.PROGS.PROGA3, described in catalog $OLD1.ACCTG
$OLD3.PROGS.PROGB3, described in catalog $OLD1.SALES
$OLD1.PROGS.PROGB1, described in catalog $OLD1.SALES
$OLD2.PROGS.PROGB2, described in catalog $OLD1.SALES
$OLD3.PROGS.PROGC3, described in catalog $OLD1.SALES
$OLD2.PROGS.PROGC2, described in catalog $OLD1.SALES
$OLD1.PROGS.PROGC1, described in catalog $OLD1.ORDERS
$OLD1.PROGS.PROGD1, described in catalog $OLD1.ORDERS
$OLD2.PROGS.PROGD2, described in catalog $OLD1.ORDERS
$OLD2.PROGS.PROGE2, described in catalog $OLD1.ORDERS
$OLD3.PROGS.PROGD3, described in catalog $OLD1.ORDERS
```

Example: Method 1 (Moving Programs With DUP)

This DUP command moves these programs to new volumes; namely, $VOL1, $VOL2, and $VOL3. The MAP NAMES option correctly defines the source and target volumes for each program. After the DUP operation, the programs are SQL compiled to register them in the new catalog.

```
>>  DUP ($OLD1.PROGS.*, $OLD2.PROGS.*, $OLD3.PROGS.*)
+>   WHERE SQLPROGRAM,
+>   MAP NAMES ($OLD1.PROGS.* TO $VOL1.PROGS.*,
+>              $OLD2.PROGS.* TO $VOL2.PROGS.*,
+>              $OLD3.PROGS.* TO $VOL3.PROGS.*), SAVEALL;
19> VOLUME $VOL1
20> SQLCOMP /IN PROGS.PROGA1, OUT $S.#HOLD/ CATALOG ACCTG
21> SQLCOMP /IN PROGS.PROGB1, OUT $S.#HOLD/ CATALOG SALES
22> SQLCOMP /IN PROGS.PROGC1, OUT $S.#HOLD/ CATALOG ORDERS
23> SQLCOMP /IN PROGS.PROGD1, OUT $S.#HOLD/ CATALOG ORDERS
24> VOLUME $VOL2
25> SQLCOMP /IN PROGS.PROGA2, OUT $S.#HOLD/ CATALOG
    $VOL1.ACCTG
26> SQLCOMP /IN PROGS.PROGB2, OUT $S.#HOLD/ CATALOG
    $VOL1.SALES
27> SQLCOMP /IN PROGS.PROGC2, OUT $S.#HOLD/ CATALOG
    $VOL1.SALES
28> SQLCOMP /IN PROGS.PROGD2, OUT $S.#HOLD/ CATALOG
    $VOL1.ORDERS
29> SQLCOMP /IN PROGS.PROGE2, OUT $S.#HOLD/ CATALOG
    $VOL1.ORDERS
30> VOLUME $VOL3
31> SQLCOMP /IN PROGS.PROGA3, OUT $S.#HOLD/ CATALOG
    $VOL1.ACCTG
32> SQLCOMP /IN PROGS.PROGB3, OUT $S.#HOLD/ CATALOG
    $VOL1.SALES
33> SQLCOMP /IN PROGS.PROGC3, OUT $S.#HOLD/ CATALOG
    $VOL1.SALES
34> SQLCOMP /IN PROGS.PROGD3, OUT $S.#HOLD/ CATALOG
    $VOL1.ORDERS
```

Example: Method 2 (Moving Programs With DUP)

This method is a multiple-step approach for moving the programs.

1. Obtain a list of the program names from the PROGRAMS table of each catalog.
2. Edit the list, specifying the program names first in DUP commands (or the corresponding OSS command for SQL programs stored in OSS files) and then in SQLCOMP (or c89, for OSS) commands to SQL compile the programs before registering them in the new catalog.

Step 1 queries the PROGRAMS table. The query produces a list of program names from the catalog $OLD1.ACCTG and writes the list in the log file $VOL1.PGMS.PROGLIST.

```
>>  LOG $VOL1.PGMS.PROGLIST;
>>  VOLUME $OLD1.ACCTG;
>>  SELECT PROGRAMNAME FROM PROGRAMS;
```

This list appears in the log file:

```
PROGRAMNAME
------------------------------
\SYS1.$OLD1.PROGS.PROGA1
\SYS1.$OLD2.PROGS.PROGA2
\SYS1.$OLD3.PROGS.PROGA3
```

Step 2 displays the edited results of the first entry in the list file. Editing has changed the listed Guardian programs into DUP commands in an OBEY command file entered at an SQLCI prompt. This step also displays the SQLCOMP commands entered through TACL to SQL compile the programs.

```
>>  DUP $OLD1.PROGS.PROGA1, $VOL1.PROGS.*, SAVEALL;
>>  DUP $OLD2.PROGS.PROGA2, $VOL2.PROGS.*, SAVEALL;
>>  DUP $OLD3.PROGS.PROGA3, $VOL3.PROGS.*, SAVEALL;
>>  EXIT
19> SQLCOMP /IN $VOL1.PROGS.PROGA1, OUT $S.#HOLD/
             CATALOG $VOL1.ACCTG
20> SQLCOMP /IN $VOL2.PROGS.PROGA2, OUT $S.#HOLD/
             CATALOG $VOL1.ACCTG
21> SQLCOMP /IN $VOL3.PROGS.PROGA3, OUT $S.#HOLD/
             CATALOG $VOL1.ACCTG
```

Steps 1 and 2 must be performed for each catalog.

11 Performing Recovery Operations

The success of recovery operations depends on the effectiveness and consistency of the plan developed for handling recovery situations. Before you begin any recovery operation, you should thoroughly evaluate the tools—backup tapes, TMF online dumps, and so forth—available and appropriate for the type of failure.

Recovery procedures described in this section use these tools:

- BACKUP and RESTORE: Guardian utilities for dumping files or tables to tape and restoring them
- Peripheral Utility Program (PUP): The Guardian utility used in D-series and earlier RVUs to manage disks and other peripheral devices, and perform various operations on disk volumes in the SQL/MP database environment. In G-series RVUs, PUP functions are performed by the SCF.
- Subsystem Control Facility (SCF): An interactive interface for configuring, controlling, and collecting information from a subsystem and its objects. SCF enables you to configure and reconfigure devices, processes, and some system variables while your HP NonStop S-series server is online.
- Volume recovery: A TMF recovery mechanism that returns a database to a consistent state after a system failure. Volume recovery reapplies committed transactions to ensure they are reflected correctly in the database and then backs out all transactions that were incomplete at the time of the interruption.
- File recovery: A TMF recovery mechanism used for recovery from disk and media failures and the effects of incorrect programs. File recovery restores the database from the most recent online dumps, applying the after-images from the audit trail to the database records, and then backs out all transactions that were incomplete at the time of the system interruption or failure.

If you need to recover a database to a different node, see Renaming or Renumbering a Node on page 9-32 in addition to the material in this section.

For information about using the RDF product to maintain a duplicate database at a remote site, see the *RDF/IMP and IMPX System Management Manual*.

Restoring Individual SQL Objects

The RESTORE utility can replace SQL objects that have been backed up on tape. For this discussion, restoring SQL objects and databases means you are replacing existing objects in the same location. For a discussion about using the RESTORE utility to move SQL objects and databases, see Section 9, Moving a Database.

RESTORE automatically creates SQL catalogs for SQL objects being restored if a catalog does not exist and the command includes the AUTOCREATECATALOG ON option.

During object restoration, if RESTORE determines that a referenced catalog does not exist, RESTORE directs the catalog manager to create the catalog, and alter the catalog security and owner ID, to match the security and owner ID at the time of the backup operation. After the catalog security and owner are altered, the user performing the restore might not have the appropriate security to update the catalog; consequently, the objects might not be restored in the catalog.

The default for RESTORE is AUTOCREATECATALOG OFF.

Restoring Catalogs

The RESTORE utility cannot directly recover a catalog. TMF recovery methods protect SQL/MP catalogs. All of the catalog tables are audited so that they can be archived by using the TMF subsystem and recovered by using either TMF volume recovery or file recovery procedures.

Restoring Collations

If you restore objects and programs that use collations, you must restore the collations first. For audited collations, however, use the TMF file recovery operation, discussed under Restoring Objects With TMF Recovery Operations on page 11-11, instead of RESTORE.

Restoring Tables

Typically, you would want to use the RESTORE utility to recover programs or nonaudited tables. For audited tables, use the TMF file recovery operation discussed under Restoring Objects With TMF Recovery Operations on page 11-11.

The RESTORE utility replaces the file with a file of the same type. You cannot use RESTORE to drop an SQL table and restore it as an Enscribe file or use it to drop an Enscribe file and restore it as an SQL table.

When restoring a table, the RESTORE utility tries to duplicate all partitions, indexes, and protection views. In addition, the utility restores all comments and constraints to the COMMENTS and CONSTRNT tables, respectively, of the target catalog. RESTORE does not attempt to restore any dependent shorthand views unless their names are explicitly included in the file set list that specifies which objects to restore.

These are options of the RESTORE utility and how they affect restoring SQL objects:

- RESTORE allows the use of a qualified file-set list to identify the source objects to be restored.
- The PURGE option effectively performs an SQL DROP statement before restoring the table. By using the default, PARTONLY OFF, with the PURGE option, you can drop all partitions of a table and all the dependencies, which are indexes, partitions, protection views, comments, and constraints; then, you can restore them all.
- The INDEXES IMPLICIT or INDEXES EXPLICIT option controls the restoration of indexes. The default is for indexes to be implicitly restored with the underlying table. If you specify INDEXES EXPLICIT, the indexes are not automatically restored with the underlying table unless specified in the file set list.

△ **Caution.** You must be extremely careful when using the INDEXES EXPLICIT option because it can cause tables to become inconsistent.

- Some of the inconsistencies or other problems that can occur by using RESTORE with the INDEXES EXPLICIT option are:
  - All components might not be restored.
  - Pointers in the file labels could point to the wrong file. (This is more likely to happen when you are moving objects with the MAP NAMES clause, however.)
  - Index data might be inconsistent with the underlying table data.
- By specifying the PARTONLY ON option, you can restore a partition of a table separately or restore multiple partitions collectively. Only partitions identified by the file set list are purged and restored.

△ **Caution.** You must be extremely careful when using the PARTONLY ON option because it can cause tables to become inconsistent.

- Some of the inconsistencies or other problems that can occur by using RESTORE with the PARTONLY option are:
  - The security of partitions might be inconsistent.
  - Recovery might commence before RESTORE PARTONLY determines whether recovery is truly viable.
  - Parallel RESTORE operations might cause deadlock.
  - Programs might be invalidated unnecessarily.
  - Physical attributes of partitions might be mismatched.
  - The definition of a recovered partition might be inconsistent with those of its associated partitions if the object definition was changed because the backup was performed.

These inconsistencies can occur because RESTORE PARTONLY uses information from the backup tape to reconstruct the definition of the recovered partition.

Steps to Restore a Table

To restore a table, perform these steps. Restoring a table invalidates dependent programs.

1. Identify the table to be restored.
2. Determine the dependencies that will be affected by the drop operation by using the DISPLAY USE OF command.
3. Enter the RESTORE command at the command interpreter prompt.
4. Determine the status of the dependencies and the validity of the programs by using the DISPLAY USE OF command or VERIFY utility.
5. SQL compile invalid programs.

Examples of Restoring Tables

This example restores a single table that has no dependencies as this command would be entered from an OBEY command file:

```
RESTORE $TAPE, $VOL1.PERSNL.EMPLOYEE, &
        CATALOG $VOL1.PERSNL, OPEN, TAPEDATE, LISTALL
```

This example restores a table with a protection view named PROTEMP and an index named XEMP. PROTEMP is registered in the same catalog and resides on the same subvolume as the table. XEMP resides on another volume and is described in catalog $VOL2.PERSNL.

```
RESTORE $TAPE, $VOL1.PERSNL.EMPLOYEE, &
        CATALOG ($VOL1.PERSNL FOR $VOL1.PERSNL.EMPLOYEE, &
                 $VOL1.PERSNL FOR $VOL1.PERSNL.PROTEMP, &
                 $VOL2.PERSNL FOR $VOL2.PERSNL.XEMP), &
        OPEN, TAPEDATE, LISTALL
```

This command restores a single partition of a table:

```
RESTORE $TAPE, $VOL1.PERSNL.EMPLOYEE, PARTONLY, &
        CATALOG $VOL1.PERSNL, OPEN, TAPEDATE, LISTALL
```

This example restores a table that has dependent indexes, but because the INDEXES EXPLICIT option is specified, the indexes are not restored automatically unless listed in the file set list. In this case, the indexes are not specified in the file set list, so only the table and protection views, if any, are restored.

```
RESTORE $TAPE, $VOL1.PERSNL.EMPLOYEE, INDEXES EXPLICIT, &
        CATALOG $VOL1.PERSNL, OPEN, TAPEDATE, LISTALL
```

This example restores a table including the dependent shorthand view, EMPSVIEW. Shorthand views are restored only when they are explicitly named in the file set list.

```
RESTORE $TAPE, ($VOL1.PERSNL.EMPLOYEE, $VOL2.ADMIN.EMPSVIEW), &
        CATALOG $VOL1.PERSNL, OPEN, TAPEDATE, LISTALL
```

The next example restores a table with a protection view from a backup operation performed on another node and volume. To restore objects from another node onto your node, you must specify the MAP NAMES option to map dependent source objects (partitions, indexes, and protection views) to target objects. You must be careful to define the MAP NAMES target file-set list correctly; if you specify an invalid mapping scheme, the complete set of source objects might not be moved to the target objects. For more information, see Choosing Utilities for the Move Operation on page 9-2.

In this example, the backup tape contains a table named EMPLOYEE (stored on the $VOL1.PERSNL subvolume) and a protection view named EMPAUX (stored on the $VOL1.PERSNAUX subvolume).

This RESTORE command illustrates how to map source objects with node names to target objects with node names and source objects without node names to target objects without node names:

```
RESTORE $TAPE, $VOL1.PERSNL.EMPLOYEE, NOUNLOAD, &
        LISTALL, OPEN, AUDITED, AUTOCREATECATALOG ON, &
        MAP NAMES (\CHI.$VOL1.PERSNL.* TO \DENV.$VOL7.PERSNL.*,
&
                               $VOL1.PERSNL.* TO       $VOL7.PERSNL.*,&
                           \CHI.$VOL1.PERSNAUX.* TO
                                                \DENV.$VOL7.PERSNAUX.*,
&
                                $VOL1.PERSNAUX.* TO
                                $VOL7.PERSNAUX.*),&
        CATALOG   ($VOL7.PERSNL FOR $VOL7.PERSNL.*, &
                   $VOL7.PERSNL FOR $VOL7.PERSNAUX.*)
```

The next example presents alternative syntax for accomplishing the tasks in the previous example:

```
RESTORE $TAPE, $VOL1.PERSNL.EMPLOYEE, NOUNLOAD, &
        LISTALL, OPEN, AUDITED, AUTOCREATECATALOG ON, &
        MAP NAMES (\CHI.$*.*.* TO \DENV.$VOL7.PERSNL.*, &
                        $*.*.* TO      $VOL7.PERSNL.*,&
                   \CHI.$*.*.* TO \DENV.$VOL7.PERSNAUX.*, &
                        $*.*.* TO      $VOL7.PERSNAUX.*),&
        CATALOG   ($VOL7.PERSNL FOR $VOL7.PERSNL.*, &
                   $VOL7.PERSNL FOR $VOL7.PERSNAUX.*)
```

Restoring Views

Protection views cannot be explicitly restored; they are restored with the underlying table only. Shorthand views, however, can be only explicitly restored.

When a table is restored, only those shorthand views explicitly identified by the file set list are automatically restored. If you use a wild-card character in a file set list and a shorthand view name satisfies the file set list, the shorthand view is considered to be explicitly identified.

Restoring a shorthand view restores any comments associated with the view. A shorthand view can be restored, but the view might subsequently be marked invalid. After restoring the complete file set list, RESTORE tries to validate the view. A shorthand view might have several underlying tables or views; RESTORE might not be able to validate the view if the underlying objects are not available. You should always check the status of the views after RESTORE completes. If a view is invalid, you must drop and re-create the view.

Views contain no physical data; therefore, you might want to re-create the view instead of performing a RESTORE of the definition.

Restoring Indexes

You should normally restore indexes automatically with the underlying table. You can prohibit the restoration of indexes by specifying INDEXES EXPLICIT in the RESTORE command so that only those indexes identified by the file set list are explicitly restored.

To restore the primary partition of an index, restore the entire index.

△ **Caution.** You must be extremely careful when using the INDEXES EXPLICIT option because it can cause tables to become inconsistent. A list of possible inconsistencies appears earlier under Restoring Tables.

Restoring Programs

Restored SQL programs are not automatically registered in an SQL catalog; the SQL sensitive flag is set off, and the programs cannot be run without first being compiled.

The RESTORE operation allows SQL programs stored in Guardian files to be explicitly SQL compiled during the restore operation. If you include the SQLCOMPILE ON option, RESTORE directs the SQL compiler to recompile the program with the DEFINEs stored in the program during the last explicit compilation.

You can use the SQLCOMPILE ON option effectively when restoring programs individually or for restoring programs when you know that the referenced SQL tables and views already exist on the system. If the referenced tables and views are not available when the program is restored, the recompile cannot produce a valid query execution plan. You should not use the SQLCOMPILE ON option if you are using RESTORE to move the database.

If you are restoring databases, file set lists, or moving objects, you might not want to use the SQLCOMPILE ON option. The recompilations can be unsuccessful if objects are restored alphabetically by volume, subvolume, and file name. If programs are restored before the tables, views, or indexes on which they depend are restored, the recompilations will be unsuccessful.

To restore a program, follow these steps:

1. Determine the name of the program and the tables or views used by the program.
2. Perform the RESTORE command.
3. If the RESTORE command uses SQLCOMPILE OFF, compile the program to validate and register the program in a catalog. You can use the STOREDDEFINES option in the SQLCOMP command if you are not restoring the programs on a different node.

This example restores a program. After being restored, the program is no longer SQL sensitive and is no longer registered in a catalog; therefore, the program must be explicitly SQL compiled.

```
RESTORE $TAPE, $VOL1.PERSNL.PROG1, TAPEDATE, LISTALL
```

This example uses the SQLCOMPILE ON option of the RESTORE command to restore and compile a program. The command specifies the catalog in which the program is registered.

```
RESTORE $TAPE, $VOL1.PERSNL.PROG1, SQLCOMPILE ON, &
        CATALOG $VOL1.PERSNL, TAPEDATE, LISTALL
```

To restore SQL programs stored in OSS files, use the appropriate OSS utility.

Restoring Databases

Restoring a complete database should be a simple process, as long as the system configuration is identical to the configuration when the backup was performed.

If you are planning to restore a complete database but include such operations as renaming the disk volumes, adding new volumes, or making other configuration changes, see Section 9, Moving a Database.

You can simplify restoring a complete database by performing certain steps before the RESTORE. If you are planning a complete database restoration for some planned event, you can accomplish the complete restoration by using only the RESTORE utility for both audited and nonaudited files and SQL objects.

Completing the Planning Phase

You should prepare for restoring the database when the database is consistent and inactive. To complete planning for restoring the database, perform these steps.

1. Ensure that the database is not active.
2. If you are not planning a volume-mode RESTORE operation, described under Restoring a Database as a Planned Event on page 11-9, create an EDIT file containing SQL statements that will re-create your catalogs. The statements must specify the same security and the same owners for each catalog and for each catalog table that can be individually secured.

   The catalog security is the security of the catalog tables, except for the USAGES, TRANSIDS, and PROGRAMS tables, which can be secured individually. In the system catalog, the CATALOGS table can also be secured individually. To find out the security and owner of your catalog tables, you can query the catalog TABLES table as follows:

   ```
   >> LOG log-file CLEAR;
   >> SELECT TABLENAME, SECURITYVECTOR, GROUPID, USERID
   +>   FROM catalog-name.TABLES
   +>   WHERE TABLENAME = "\system.$volume.catalog-name.TABLES";
   ```

   To find out the security and owner of the USAGES, TRANSIDS, and PROGRAMS tables, you can specify those tables in a query:

   ```
   >> SELECT TABLENAME, SECURITYVECTOR, GROUPID, USERID
   +>   FROM catalog-name.TABLES
   +>   WHERE TABLENAME = "\system.$volume.catalog-name.USAGES"
   +>   OR TABLENAME = "\system.$volume.catalog-name.TRANSIDS"
   +>   OR TABLENAME = "\system.$volume.catalog-name.PROGRAMS";
   ```

   The EDIT file must contain these two types of statements:

   - A CREATE CATALOG statement for each catalog on the node. To make sure the catalog is re-created with the same security, use the SECURE option to specify the catalog security:

     ```
     >> CREATE CATALOG $volume.subvolume SECURE "security-
     string";
     ```

   - An ALTER TABLE statement for any table whose security or owner is different from the catalog security or owner:

     ```
     >> ALTER TABLE $volume.subvolume.PROGRAMS
     +>   SECURE "security-string"
     +>   OWNER "group-num, user-num" ;
     ```

3. Back up the entire node and use the AUDITED option so that both audited and nonaudited files are dumped. To back up the node, enter:

   ```
   BACKUP $TAPE, *.*.*, OPEN, AUDITED, LISTALL
   ```

4. If you need a list of programs that will be invalidated by this procedure, use these commands to produce a list of programs in a log file:

```
>>  LOG log-file;
>>  FILEINFO *.*.* WHERE SQLPROGRAM;
```

Restoring a Database as a Planned Event

To restore a database as a planned event, follow these steps:

1. Check that SQL is running on the node. The system catalog must be present, and the TMF subsystem must be running. If you need to reinstall the SQL system, perform the installation as described under Reinstalling SQL/MP Software on page 2-10.
2. If you are not performing a volume-mode backup, re-create the catalogs. Use the SQLCI OBEY command to run the statements in the EDIT file you created before the BACKUP operation.

   The catalog owner is the user ID executing the CREATE CATALOG statement. Ownership can later be given to another user ID, if necessary, by using the ALTER CATALOG statement.
3. Issue the RESTORE command.

   For a file-mode RESTORE operation, use a set of commands that match those used in the BACKUP process. This is an example of the RESTORE command:

   ```
   RESTORE $TAPE, *.*.*, AUDITED, OPEN, LISTALL, TAPEDATE
   ```

△ **Caution.** Do not use the SQLCOMPILE option in the RESTORE command with this type of restoration. The program compilations could cause invalid programs because dependent tables, views, and indexes might not yet be restored when the program is restored.

   To restore SQL programs stored in OSS files, use the appropriate OSS utility.

4. After you have restored all the tables and views that the programs use, SQL compile the programs.

   For a volume-mode RESTORE operation, issue a command like this for each disk volume containing database files:

   ```
   RESTORE $TAPE, VOLUMEMODE, *
   ```

△ **Caution.** Use volume-mode RESTORE only if the configuration of the restored disk is identical to the configuration of the backed up disk. If you change the configuration, you could lose a volume of data.

5. SQL compile the programs. A list of programs is saved in a log file created in Step 3 of the planning activities.

6. Verify the database by using the VERIFY utility; following is an example of the VERIFY command:

   ```
   >>  VERIFY *.*.*;
   ```

7. Drop and re-create any invalid shorthand views. By using VERIFY in Step 6, you can identify any invalid shorthand views.
8. Perform new TMF online dumps of all catalogs and audited SQL objects.

Restoring a Database as an Unplanned Event

If you were not able to plan for restoring the database because of a catastrophic failure, you must begin to restore the system by using the most recent backups and TMF online dumps.

To restore the database as an unplanned event:

1. Perform TMF file recovery for all audited files as described under Restoring Objects With TMF Recovery Operations on page 11-11.
2. Issue the RESTORE command to recover Enscribe and nonaudited files. Use a set of commands that match those used in the BACKUP process. Also include the AUTOCREATECATALOG ON option in the RESTORE command to create the necessary catalogs. An example of the RESTORE command, where *.*.* represents unaudited and Enscribe files, follows:

   ```
   RESTORE $TAPE, *.*.*, AUTOCREATECATALOG ON, OPEN, &
           LISTALL, TAPEDATE
   ```

   You should not use the SQLCOMPILE option in the RESTORE command with this type of restoration. The program compilations could cause invalid programs because dependent tables, views, and indexes might not yet be restored when the program is restored.

   To restore OSS files, use the appropriate OSS utility.

3. After you have restored all the tables and views that the programs use, SQL compile the programs.
4. Manually resolve any inconsistencies in the data between audited and nonaudited tables. At this step, your database might be consistently defined in the catalogs, but the data in the audited and nonaudited files might be inconsistent. The inconsistency occurs from the time difference between the backups and the online dumps.
5. Verify the database by using the VERIFY utility; enter:

   ```
   >>  VERIFY *.*.*;
   ```

6. Drop and re-create any invalid shorthand views. Using VERIFY in Step 5 identifies any invalid shorthand views.
7. Perform new TMF online dumps of all catalogs and audited SQL objects.

Recovering Consistent Files by Resetting the BROKEN Flag

When a disk volume or node crashes or a process terminates unexpectedly, files that are open at that time are left in a questionable state. In many cases, the files are really inconsistent because they were actively involved in interrupted database transactions. These files must be recovered with the volume recovery or file recovery methods. In other cases, files marked as questionable are actually consistent. These files, although open at the time of the crash, were not actively taking part in database transactions.

In many cases, you know which files are actually corrupt and which are actually consistent. Normally, it is better to allow TMF recovery to recover all the files and to determine which are corrupt and which are not. If, however, you are able to determine that a file is not corrupt, it can be much quicker to simply reset the BROKEN flag that indicates to the system that the file is corrupt. To reset this flag, issue an ALTER TABLE or ALTER INDEX statement using the RESETBROKEN option.

Use the RESETBROKEN option to reset the BROKEN flags for SQL catalog tables. A catalog, although sometimes open at the time of a crash (as a result of activities such as automatic recompilation or dynamic SQL operations), is often not actively involved in update operations. Also, to facilitate recovery of database files, you can reset the BROKEN flags for the catalog tables if these tables are not corrupt.

You must use the RESETBROKEN option before you use a TMF recovery method. After it starts, TMF recovery resets the flag.

△ **Caution.** Avoid using the RESETBROKEN option on files that might be corrupt. If you are unsure of the state of a file, use TMF recovery methods instead. RESETBROKEN is not a replacement for other recovery methods when the file is corrupt.

Restoring Objects With TMF Recovery Operations

With the TMF volume recovery and file recovery mechanisms, you can recover SQL catalogs and objects after a system or disk volume crash. You can also use file recovery to recover a purged object and to recover a database to a specified time. These recovery operations, and others, are described next. For additional information about volume recovery and file recovery, see The TMF Subsystem on page 4-10 or the *TMF Operations and Recovery Guide*.

Database Recovery After a Disk or Node (System) Failure

When a disk or node (system) fails, often SQL catalogs tables and database files on the disk or node are left in a crash-open state. To recover the database, both the catalogs and the files must be recovered to a consistent state.

Depending upon the situation, choose an appropriate method from this list to achieve the desired result:

- Use the file recovery method to recover the database, starting with online dumps (files containing copies of consistent catalogs and objects saved by the TMF DUMP FILES command). The file recovery function starts with the saved files and updates transactions to the last consistent point in the audit trails.
- Use the file recovery method with a specified TIME option to recover a database to a given consistent time, as described under File Recovery With the TIME Option on page 11-14.
- Use the file recovery method to recover files that cannot be recovered by volume recovery because the audit trails are missing or damaged in some way. In some cases, the damage could also prevent file recovery to the most recent point.

Volume Recovery

TMF volume recovery is invoked automatically by the TMF commands START TMF, and is invoked as needed thereafter when a volume becomes accessible. Volume recovery uses the audit trails to roll back incomplete transactions and return the database to the last consistent state.

Volume recovery might fail to recover a volume or a file. Some of the recoverable cases follow:

- A volume becomes unavailable during the volume recovery operation. When you bring up the volume, TMF automatically restarts volume recovery to the last recovery point in the database.
- A file is corrupted or inconsistent in such a way that volume recovery cannot apply the audit trail information. If volume recovery fails to recover a file, FILEINFO displays setting of the REDONEEDED and UNDONEEDED flags. For tables, indexes, and Enscribe files, the information appears after the modification timestamp of the table. For views, the information appears after the open states LABEL QUESTIONABLE and DEFINITION INVALID if they appear in the display.

  Normally, volume recovery recovers such files when the volume is started for transaction processing. If, however, the volume is already started and the file is still marked with REDONEEDED or UNDONEEDED, you must recover the file by using file recovery.

File Recovery

File recovery is usually the recovery method used if other methods have failed. File recovery can be used only if you consistently dump audit trails to tape and make online dumps. File recovery reconstructs an audited file from the initial starting point of the online dumps and applies all the changes to the file from the history of the audit trails. The file is recovered to the last consistent point in the database. These guidelines apply:

- The file recovery process is invoked by issuing the RECOVER FILES command to one of the TMF interfaces (such as TMFCOM). The file recovery process prompts the operator for the online dumps and audit-trail tapes as needed. Audit trails that still reside on disk are read directly from disk.
- If you do not specify the FROMARCHIVE option in the RECOVER FILES command, the file recovery process recovers only the files marked undo-needed. If you specify the FROM ARCHIVE option of the RECOVER FILES command, file recovery tries to recover the entire file set, regardless of the setting of the redo-needed and undo-needed flags.
- The file recovery process cannot recover any file that did not exist at the time of an online dump. The file recovery process cannot perform a create function. You must perform an online dump following any create operation. If you do not perform this dump, you cannot recover the file because the TMF subsystem looks for a starting point on the most recent online dump.
- If your database uses a scheme of audited and nonaudited files, you might not be able to recover a consistent database, depending on the date and time of the most recent BACKUP and TMF recovery point of audited files. You must then manually attempt to put the database into a consistent state.
- A REDONEEDED or UNDONEEDED flag in the FILEINFO display for a file indicates that you must use file recovery to recover the file.
- If your system uses the SMF product to manage disk volumes, TMF file recovery procedures might differ slightly from those described in these sections. For example, a file might be recovered to a different disk volume if the volume on which it originally resided is not available. For more information about how TMF performs file recovery on volumes managed by SMF, see TMF manuals. For more information about SMF, see the *Storage Management Foundation User’s Guide*.

File Recovery With the TIME Option

By using the file recovery feature with the TIME option, you can resolve several different kinds of problems:

- If a database object is purged by accident, you can use the TIME option to recover the object's file as it existed just prior to the purge. This action effectively recovers the entire file but not the catalog definition of the object.
- If an application error updates the database in an inconsistent way, you can recover the database to the state it was in at a specified time before the error occurred.
- If a licensed SQLCI2 or CLEANUP operation incorrectly alters or damages the database or catalogs, you can recover the database or catalogs to their previous state.
- If you have a saved test database or starting database, you can recover that database to the same point many times. Suppose that in your testing procedures you need to always start with the same database. This database can be loaded to the node or recovered by using TMF file recovery with the TIME option.

Using file recovery with the TIME option can be difficult, however, because this method requires you to coordinate recovery of interrelated objects, such as tables and their indexes.

△ **Caution.** The TMF subsystem carries no information about the relationships between file labels and catalogs. If a table is dropped, for example, file recovery cannot restore the catalog entries for the table. If the file recovery operation starts at a time just before a table was dropped, you might lose subsequent DDL changes.

For more information about using file recovery with the TIME option, see the *TMF Operations and Recovery Guide*.

Recovering Purged SQL Tables

There are two ways of using TMF to recover an SQL table that was accidentally purged:

- Use file recovery to recover the catalog and the purged table. This approach works adequately only if no updates were applied to the catalog after the table was purged (which is usually not the case).
- Re-create the table to put the entry back into the catalog, and then recover the table and update the creation and redefinition timestamps in the catalog (if needed).

For detailed steps to recover accidentally dropped tables, see Recovering Tables on page 11-19.

Note. If you follow the first approach, and any dependant objects of the dropped table were registered in different catalog(s), those catalogs must also be recovered along with the catalog in which the dropped table was registered.

Operations That Invalidate TMF Online Dumps

Some SQL/MP operations invalidate TMF online dumps, affecting TMF file recovery. The TMF subsystem maintains the integrity and consistency of databases for online transaction processing. You must understand how SQL/MP and the TMF subsystem work together so that you do not lose or damage important data.

To execute any SQL/MP operation that invalidates online dumps, you must have either the super ID or ownership of all affected tables.

Some SQL/MP operations invalidate TMF catalog entries, which invalidates TMF online dumps. These SQL/MP operations delete or significantly alter the file labels or the file contents.

△ **Caution.** If the TMF catalog entries are incorrect and a problem occurs with the database, you could lose the ability to use TMF file recovery operations to recover the database.

To keep file recovery protection for these files, you must make new TMF online dumps after any of these operations. Even if the operation fails to complete properly, file labels or file contents might be affected. Plan to make new TMF online dumps even if one of these operations is unsuccessful.

If you need to recover an affected table or index to a point before the SQL operation that invalidated the applicable dump, the TMF file recovery process might require that you manually modify the online and audit dump entries in the TMF catalog by using the TMF ALTER DUMPS or ADD DUMPS command. To preserve consistency, this type of a recovery must include not only the tables or indexes directly affected, but also all partitions of each table or index and all logically related objects in the database.

△ **Caution.** If a full recovery of a table is needed and the catalog is not going to be recovered, the timestamps can cause inconsistencies that leave the table unusable.

For more information on making online dumps, see the *TMF Operations and Recovery Guide.*

Table 11-1. SQL/MP Operations That Invalidate TMF Online Dumps

SQL Statement	Option	Effect	Recovery Strategy
ALTER INDEX and ALTER TABLE	NO AUDIT	Invalidates all online dumps of the affected object. The object does not have any TMF file recovery protection if it is not audited.	If the AUDIT attribute is later turned back on, make new online dumps of all partitions of the index or table to retain TMF file recovery protection.
	ADD PARTITION (with data movement)	Invalidates all online dumps of the source partition.	Make new online dumps of the source partition and added partition to retain TMF file recovery protection.
	MOVE (simple move)	Invalidates all online dumps of the source partition.	Make a new online dump of the moved partition to retain TMF file recovery protection.
	MOVE (one-way split)	Invalidates all online dumps of the source partition.	Make new online dumps of the source partition and moved partition to retain TMF file recovery protection.
	MOVE (two-way split)	Invalidates all online dumps of the source partition.	Make online dumps of the new partitions to provide TMF file recovery protection.
	RENAME	Invalidates all online dumps of the renamed object.	Make a new online dump of the renamed object to keep TMF file recovery protection for it.

Responding to Accidental Loss of an Audited SQL/MP Object

The method for recovering an accidentally dropped SQL object depends on whether that object is a view, an index, or a table.

Recovery of a single view or index is usually a straightforward operation. Recovery of a table, however, can be complex and difficult, particularly if the table has multiple dependent objects. For safety's sake, take the precautions discussed next to prevent accidental loss of an object or to simplify recovery if it does become necessary.

Recovery Precautions

- Set the NOPURGEUNTIL attribute for your objects to some date in the far future, using the SQLCI ALTER command. For example, this SQLCI command sets the NOPURGEUNTIL attribute for the table named $DATA.PERSNL.EMPLOYEE to a safe date. (If this is a partitioned table, this command sets NOPURGEUNTIL for all partitions.)

  ```
  >>ALTER TABLE $DATA.PERSNL.EMPLOYEE NOPURGEUNTIL DEC 31 2050;
  ```

  If you later try to purge the object before the NOPURGEUNTIL date, the purge fails, and you receive an error message. Now, the only way you can remove the object is to change the NOPURGEUNTIL date and then retry the purge.
- Maintain current OBEY command files containing SQLCI command scripts for creating and re-creating your SQL tables, indexes, and views.

Note. Because SQLCI does not provide an OBEYFORM option, you must manually create the OBEY command files in edit format, using your text editor.

- If you alter an object, be sure to alter the OBEY command file used to create that object too.
- Anytime you perform a SQLCI DDL operation, also request a TMF online dump for the affected object. (With each new dump, you decrease the number of tapes that must be processed during future recovery operations.)
- Maintain a hard copy of the entire TMF catalog, using the TMFCOM INFO DUMPS, DETAIL command.
- Whenever you request a TMF online dump, back up that dump to tape and use the TMFCOM INFO DUMPS, OBEYFORM command to obtain a hard copy with that tape. For good TMF practice, be sure to maintain a backup copy of the entire TMF catalog on tape.
  - If you perform the dumps with separate groups of disks (for example, a dump for each group attached to a particular processor), the dumps for your SQL objects and catalogs will be scattered among numerous tapes. The advantage of this approach is that you are less likely to miss a vital object during recovery.

The disadvantage is that it requires a lot of work because you must continually keep track of all interdependent objects and process many tapes during recovery.

- If you perform collective dumps of the SQL catalog and all its objects, you might gain a faster recovery, but you must continually update the SQL OBEY command files that you use to rebuild your SQL objects.

Note. You can use OBEY command files containing TMFCOM command scripts for TMF tasks that you perform repeatedly.

Finally, before dropping an object, check that you have:

- A current OBEY command file for re-creating your objects
- Output from a SQLCI DISPLAY USE OF command, showing for each object the other objects that depend upon it
- Current online dumps of the objects
- Hard copies of the TMF catalog and object dumps, obtained with the TMFCOM INFO DUMPS, OBEYFORM command

Recovering Views and Indexes

If the SQL object purged is a view or an index and its related table still exists in the system, you can recover the object by simply re-creating it:

- An SQL view does not contain data. The data referenced by the view is stored in the underlying table. Therefore, you can easily return a purged view to the database by re-creating the view definition using the SQLCI CREATE VIEW statement. After you recover the view, be sure to make a new online dump of the view and its related table.
- An SQL index specifies an alternate access path to a table. You can recover a purged index by re-creating it using the SQLCI CREATE INDEX statement. This approach ensures that the new index includes keys for all rows of the table. After you recover the index, make a new online dump of the index and its related table.

You can also recover a view or an index by using the TMFCOM RECOVER FILES command, using the steps described under Recovering Tables on page 11-19. However, because it is potentially more complex and open to error, do not use the TMFCOM RECOVER FILES command if you are attempting to recover views and indexes only. To recover only views or indexes, use the SQLCI CREATE statements discussed previously.

Recovering Tables

If the SQL object purged is a table, recovery can be much more complex than one involving only views and indexes. In some cases (for example if the TMF subsystem is not configured for file recovery), recovery might not be possible at all. For this reason, follow the Recovery Precautions on page 11-17.

△ **Caution.** Unless performed with great care and precision, SQL table recovery involves risk of database corruption and loss of data integrity. Recovery should be done only by experienced users of SQL/MP and TMF users who understand:

- How objects are defined in the SQL catalog and the results of altering those definitions
- How to use the licensed SQLCI2 utility

If no one with this expertise is present at your site, contact the Global Customer Support Center (GCSC) or your service provider before proceeding.

For the best results in most cases, to recover a table and its dependent objects:

1. Determine what dependent objects (views, indexes, and other tables) might have been dropped along with the table, using SQLCI.
2. Re-create the table and its dependent objects, using SQLCI. The DDL definition of the newly created table must exactly match the DDL definition of the purged table.
3. Reset the INVALID and RELEASED attributes of the online dumps for the dropped objects to OFF, using the TMFCOM ALTER DUMPS command.
4. If any indexes were associated with the file, re-create them.
5. Recover the table and its dependent objects with the TMFCOM RECOVER FILES command, using the TOFIRSTPURGE option.
6. For the recovered objects, verify that the creation and redefinition timestamps in the file labels match those in the SQL catalog, using the SQLCI VERIFY command.
7. For all objects for which VERIFY identifies a mismatch, update the timestamps in the SQL catalog to match those in the file labels, using a licensed SQLCI2 utility.
8. Update the statistics for the recovered table.
9. SQL compile any SQL programs that access this table.

Partitioned Tables

Recovery of partitioned tables requires special attention. The CREATE statement for a partitioned table must indicate the number and names of the partitions as they were at the time the table was dropped. (If you are recovering tables to a different location, however, their partition names can be different.)

Over time, partitions are dropped, moved, added, and split. To rebuild a CREATE statement reflecting the partitions at the time of the drop, therefore, you should file a copy of a FILEINFO, DETAIL statement for partitioned files, together with a copy of an INVOKE statement, after the most recent change.

After you have re-created the partitions and recovered them with TMF, the timestamps in the catalogs might be wrong for every partition. Because the redefinition timestamp is the same for all partitions, you can use a single UPDATE statement for each catalog involved.

If a mismatch is identified, however, you must update the creation timestamp individually. Use care when updating because earlier versions of VERIFY do not name the partitions having the mismatched timestamp. If you are using an earlier version, follow the method described in Step 8 of the Recovery Example.

Recovery Example

For example, suppose that you have defined a table named EMPLOYEE on the subvolume \HIL3.$DATA.PERSNL. A SQLCI DISPLAY USE OF command lists the EMPLOYEE table and its dependent objects: EMPLIST (a protection view), MGRLIST and ORDREP (two shorthand views), and XEMPDEPT and XEMPNAME (two indexes):

```
>>DISPLAY USE OF $DATA.PERSNL.EMPLOYEE;

  Object Name                    Type S P  Owner Name   Secure
  --------------------------     ---- - - -----------   ------
         Catalog Name
         ---------------------

  0 \HIL3.$DATA.PERSNL.EMPLOYEE TA        TEG   .SAM   GG00
        $DATA.PERSNL
  1 \HIL3.$DATA.PERSNL.EMPLIST  PV        TEG   .SAM   GG00
        $DATA.PERSNL
  1 \HIL3.$DATA.PERSNL.MGRLIST  SV        TEG   .SAM   GG00
        $DATA.PERSNL
  1 \HIL3.$DATA.PERSNL.XEMPDEPT IN        TEG   .SAM   GG00
        $DATA.PERSNL
  1 \HIL3.$DATA.PERSNL.XEMPNAME IN        TEG   .SAM   GG00
        $DATA.PERSNL
  1 \HIL3.$DATA.SALES.ORDREP    SV        TEG   .SAM   GG00
        $DATA.SALES

  U = Undefined node    N = Node unavailable     T = Unsupported type
  @ = Node not in list  * = Previously displayed ? = System error

  Number of unique dependencies : 5
  Number of direct dependencies : 5
```

Later, you discover that someone has issued a SQLCI DROP TABLE command that purged the EMPLOYEE table:

```
>>DROP TABLE $DATA.PERSNL.EMPLOYEE;
---SQL operation complete.
```

You determine that this table was dropped inadvertently. To recover it:

1. Verify that the EMPLOYEE table has been removed from the database by entering the SQLCI DISPLAY USE OF command:

```
>> DISPLAY USE OF EMPLOYEE;
*** ERROR from SQL [-1220]: The label of \HIL3.$DATA.PERSNL.EMPLOYEE
***      could not be accessed.
*** ERROR from File System [11]:  file not in directory or row not
***      in file, or the specified tape file is not present on a
***      labeled tape.
>>
```

These error messages confirm that the table has been removed.

2. Identify the EMPLOYEE table's dependent objects that might also have been dropped. To do this, check the output from the last DISPLAY USE OF command issued for this table (see Recovery Example on page 11-20) to determine what objects depend on the table. Now you can conclude that the EMPLIST protection view, the MGRLIST and ORDREP shorthand views, and the XEMPDEPT and XEMPNAME indexes have also been dropped.

3. Further confirm that these dependent objects were actually dropped by issuing a TMFCOM INFO DUMPS, DETAIL command for each object. These commands list the online dump entries in the TMF catalog for the objects. The dump entries for three objects (EMPLOYEE, EMPLIST, and MGRLIST) appear next:

```
~INFO DUMPS $DATA.PERSNL.EMPLOYEE, DETAIL

                       Dump                    Dump
      Date-Time        Type   Master Data   Status
---------------------  -----  ------ -----  -------
12-Dec-1997  14:15:12  online 1      1      invalid. . .
                      .
                      .
                      .
 ~INFO DUMPS $DATA.PERSNL.EMPLIST, DETAIL

                       Dump
       Date-Time       Type   Master Data   Status
---------------------  -----  ------ -----  -------
12-Dec-1997  14:15:14  online 1      1      invalid. . .
                      .
                      .
                      .
 ~INFO DUMPS $DATA.PERSNL.MGRLIST, DETAIL

                       Dump
       Date-Time       Type   Master Data   Status
---------------------  ----   ------ -----  -------
12-Dec-1997  14:15:17  online 1      1      invalid. . .
                      .
                      .
                      .
```

The dump status “invalid” that appears for each object indicates that the object was lost.

Similar INFO DUMPS DETAIL commands for XEMPNAME, XEMPDEPT, and ORDREP reveal that they, too, were dropped.

4. Re-create the EMPLOYEE table and all its dependent objects:

   a. Check that the OBEY (script) command file you maintain for this purpose contains a SQLCI CREATE command for the EMPLOYEE table and all its dependent objects.

   b. Issue the SQLCI OBEY command to execute the commands in the OBEY command file. (In this case, the OBEY command file is named DBCREATE.)

   ```
   >>OBEY DBCREATE
   ```

   As SQLCI executes the commands, they appear onscreen, along with certain related messages:

```
>>?SECTION employee
>>  CREATE TABLE =employee (
+>                         empnum        NUMERIC (4) UNSIGNED
+>                                       NO DEFAULT
+>                                       NOT NULL
+>                                       HEADING "Employee/Number"
+>                        ,first_name    CHARACTER (15)
+>                                       DEFAULT SYSTEM
+>                                       NOT NULL
+>                                       HEADING "First Name"
+>                        ,last_name     CHARACTER (20)
+>                                       DEFAULT SYSTEM
+>                                       NOT NULL
+>                                       HEADING "Last Name"
+>                        ,deptnum       NUMERIC (4)
+>.                                      UNSIGNED
+>                                       NO DEFAULT
+>                                       NOT NULL
+>                                       HEADING "Dept/Num"
+>                        ,jobcode       NUMERIC (4) UNSIGNED
+>                                       DEFAULT NULL
+>                                       HEADING "Job/Code"
+>                        ,salary        NUMERIC (8, 2) UNSIGNED
+>                                       DEFAULT NULL
+>                                       HEADING "salary"
+>                        ,PRIMARY KEY (empnum)
+>                        )
+>  CATALOG =persnl
+>  ORGANIZATION KEY SEQUENCED
+>  ;
--- SQL operation complete.

>>  CREATE VIEW =EMPLIST
+>    AS SELECT
+>         empnum
+>        ,first_name
+>        ,last_name
+>        ,deptnum
+>        ,jobcode
+>       FROM =employee
+>    FOR PROTECTION
+>    CATALOG =persnl
+>  ;
--- SQL operation complete.
                    .
                    .
                    .
>>  CREATE INDEX =xempname
+>    ON =employee (
+>                  last_name
```

```
+>                  ,first_name
+>                  )
+>  CATALOG =persnl
+>  ;
--- SQL operation complete.
                     .
                     .
                     .
>> CREATE INDEX =xempdept
+>   ON =employee (
+>                  deptnum
+>                 )
+>   CATALOG =persnl
+>  ;
--- SQL operation complete.
                     .
                     .
                     .
>>  CREATE VIEW =mgrlist (
+>                         first_name
+>                        ,last_name
+>                        ,department
+>                        )
+>    AS SELECT
+>         first_name
+>        ,last_name
+>        ,deptname
+>       FROM
+>         =dept
+>        ,=employee
+>       WHERE
+>         dept.manager = employee.empnum
+>   CATALOG =persnl
+>  ;
--- SQL operation complete.
                     .
                     .
                     .
>>  CREATE VIEW =ordrep
+>    AS SELECT empnum
+>             ,last_name
+>             ,ordernum
+>             ,o.custnum
+>       FROM
+>         =employee e
+>        ,=orders o
+>        ,=customer c
+>       WHERE
+>        e.empnum = o.salesrep
+>       AND
+>       o.custnum = C.custnum
+>   CATALOG =sales
+>  ;
--- SQL operation complete.
                     .
                     .
                     .
```

5. Verify that all the objects have been re-created by issuing the SQLCI DISPLAY USE OF command:

```
>> DISPLAY USE OF $DATA.PERSNL.EMPLOYEE;
```

The resulting display is identical to the one shown under Recovery Example on page 11-20. The objects exist once again. However, the EMPLOYEE table does not yet contain any data.

6. When the objects were purged, TMF set the INVALID and RELEASED attributes of the online dumps for the objects to ON. Before you can recover the objects, you must first reset these attributes to OFF, using the TMFCOM ALTER DUMPS command:

```
~ ALTER DUMPS (                      &
~              $DATA.PERSNL.EMPLOYEE &
~             ,$DATA.PERSNL.EMPLIST  &
~             ,$DATA.PERSNL.MGRLIST  &
~             ,$DATA.PERSNL.XEMPNAME &
~             ,$DATA.PERSNL.XEMPDEPT &
~             ,$DATA.SALES.ORDREP    &
~             )                      &
~             ,INVALID OFF           &
~             ,RELEASED OFF          &
~             ,SERIAL 73
```

Note. If, for any reason, the dumps were completely removed from the TMF catalog, you would need to add them again, using the TMFCOM ADD DUMPS command. In this command, you would also set the INVALID and RELEASED attributes to OFF.

Now, you are ready to recover the table and its dependent objects.

△ **Caution.** Normally, the objects to be recovered are spread across different disk volumes and subvolumes. However, in this example, some of the objects are located in the same subvolume as the SQL/MP catalog. In such a case, use care to avoid recovering the catalog tables so that the current state of the catalog is maintained.

7. Proceed with recovery by entering the TMFCOM RECOVER FILES command:

```
~ RECOVER FILES (                      &
~                $DATA.PERSNL.EMPLOYEE &
~               ,$DATA.PERSNL.EMPLIST  &
~               ,$DATA.PERSNL.MGRLIST  &
~               ,$DATA.PERSNL.XEMPNAME &
~               ,$DATA.PERSNL.XEMPDEPT &
~               ,$DATA.SALES.ORDREP    &
~               )                      &
~               ,FROMARCHIVE           &
~               ,TOFIRSTPURGE
```

The objects are now recovered in the database, but additional work might remain to be done because of possible inconsistencies between the objects' file labels on disk and the corresponding information for them in the SQL catalog.

TMF does not update or insert entries in the SQL catalog during a RECOVER FILES operation for a table. As it performs recovery, TMF attempts to automatically synchronize the objects' create time and redefinition time between the catalog and the file label on disk. If TMF cannot perform this synchronization, it displays EMS Message 203 (with Subsystem Error 9038) for each inconsistency. For example:

```
NonStop TMF on \PLUTO *0203* RECOVER FILES [57]
OnLineRestore Process #1 OnlineDumpMgmt: *WARNING*
TMF-9038: $DATA.PERSNL.EMPLIST: Unable to retrieve the
CreateTime and RedefTime for this object from disk. Using the
values from the online dump instead.
```

8. At the end of recovery operation, use these EMS messages to determine which files have timestamps on disk that are inconsistent with their timestamps in the SQL catalog. You must then manually perform the synchronization for these objects, as explained in Step 9 on page 11-26.

   Alternatively, use the SQLCI VERIFY utility to list the inconsistencies between the object descriptions in the file labels and in the SQL catalog:

```
>>VERIFY $DATA.PERSNL.*;
                       .
                       .
                       .
--- Verifying $DATA.PERSNL.EMPLIST
*** ERROR from SQL [-9853]: Column LA^CrTime^F in disk label does
***      not match TABLES.CreateTime in catalog \HIL3.$DATA.PERSNL.
*** ERROR from SQL [-9886]: Value of LA^CrTime^F is:
***      211929379570628303 in partition \HIL3.$DATA.PERSNL.EMPLIST
*** ERROR from SQL [-9853]: Column CatalogOptime^F in disk label
***      does not match TABLES.Redeftime in catalog
***      \HIL3.$DATA.PERSNL.
*** ERROR from SQL [-9886]: Value of CatalogOptime^F is:
          211929379552916402 in partition \HIL3.$DATA.PERSNL.EMPLIST
                       .
--- Verifying $DATA.PERSNL.EMPLOYEE
--- $DATA.PERSNL.EMPLOYEE verified.
                       .
                       .
                       .
--- Verifying $DATA.PERSNL.MGRLIST
--- $DATA.PERSNL.MGRLIST verified.
                       .
                       .
                       .
--- Verifying $DATA.PERSNL.XEMPDEPT
-- $DATA.PERSNL.XEMPDEPT verified.
                       .
                       .
                       .
--- Verifying $DATA.PERSNL.XEMPNAME
-- $DATA.PERSNL.XEMPNAME verified.
                       .
                       .
                       .

>>VERIFY $DATA.SALES.*;
                       .
                       .
                       .
--- Verifying $DATA.SALES.ORDREP
-- $DATA.SALES.ORDREP verified.
                       .
                       .
                       .
--- SQL operation complete.
```

If no objects are identified as inconsistent, recovery is complete. Otherwise, proceed to Step 9 on page 11-26.

9. Use a licensed SQLCI2 utility to update the timestamps in the SQL catalog to match those in the file labels for all objects identified as inconsistent in Step 8. Be sure to use a log file to record the changes to be made. Also, to reduce error, HP recommends that you use fully qualified object names in the commands you enter. These commands create a log file and accomplish this updating for the objects identified in Step 8 on page 11-25 :

```
>>LOG $DATA.FIXUP.FIX1;
>>UPDATE $DATA.PERSNL.TABLES SET CREATETIME =
211929379570628303
>+WHERE TABLENAME = "\HIL3.$DATA.PERSNL.EMPLIST";
--- 1 ROW(S) UPDATED.

>>UPDATE $DATA.PERSNL.TABLES SET REDEFTIME =
211929379552916402
>+WHERE TABLENAME = "\HIL3.$DATA.PERSNL.EMPLIST";
--- 1 ROW(S) UPDATED.
```

10. Use the SQLCI VERIFY utility once again, as in Step 8 on page 11-25, to validate the entries in the catalog against those in the file labels. If you find no mismatches, you know that recovery of the table and its dependent objects is complete and your work is done. Otherwise, return to Step 9.

```
>>VERIFY $DATA.PERSNL.*;
--- Verifying $DATA.PERSNL.EMPLIST
--- $DATA.PERSNL.EMPLIST verified.
                  .
                  .
                  .
```

11. Use the UPDATE ALL STATISTICS command to update the table statistics:

```
>>UPDATE ALL STATISTICS for TABLE $data.persnl.employee;

--- SQL operation complete.
```

12. Determine if any SQL programs access the recovered table and do an explicit SQL recompilation for those programs using SQLCOMP.

Tables That Have Indexes

Always remember that if a table has indexes, it is better to re-create the indexes, along with the table, and then to recover them, along with the table, in the same TMFCOM RECOVER FILES command. Otherwise, recovery will face even greater problems. The number of indexes is maintained in the file label in the disk directory. When you use SQLCI to create just the table and not the indexes, and later recover the table, additional mismatches will occur between the SQL catalog and the file label in the directory, making recovery even more difficult.

Recovering Files to New Volumes, Subvolumes, or File-IDs

You can also re-create SQL objects under different file IDs and place them on different volumes or subvolumes than the source objects. Target objects can be created in a different SQL catalog, but the target object description in the catalog must match that in the source catalog. For example, in the case of a partitioned file, both the target and source files must have the same number of partitions. Indexes for the target and source files must match.

Because TMF does not apply SQL file-label modification records encountered in an audit trail for a source object being recovered to the target object, the file label in the online dump must match the file label of the newly created target, and the target's file label must match the final form of the source file label. Therefore, to recover to a new location, you must take new online dumps each time the file label is modified.

△ **Caution.** Do not attempt to recover SQL catalog files to a new location, because this action creates unusable SQL objects.

SQL objects being recovered to a new location must be created before recovery, and the target objects must exactly match the source objects. If you attempt to recover without creating the target object, the restore process fails with Error 9037, as shown in this example, and the object is not recovered:

```
NonStop TMF on \PLUTO *0202* RECOVER FILES [58]
OnLineRestore Process #1 OnlineDumpMgmt: *ERROR*
TMF-9037: $DATA17.PERSNL.EMPLOYEE: File System error 11
occurred attempting to retrieve the SQL file label from
disk.
```

If, for any other reason, the target object is inaccessible during the restore process, this process also fails with Error 9037. For example:

```
NonStop TMF on \PLUTO *0202* RECOVER FILES [58]
OnLineRestore Process #1 OnlineDumpMgmt: *ERROR*
TMF-9037: $DATA17.PERSNL.EMPLIST: File System error 1059
occurred attempting to retrieve the SQL file label from
disk.
```

If TMF detects a mismatch between the source file's label and the target file's label, the object's recovery fails with Error 9036. For example:

```
NonStop TMF on \PLUTO *0202* RECOVER FILES [59]
OnLineRestore Process #1 OnlineDumpMgmt: *ERROR*
TMF-9036: $DATA17.PERSNL.EMPLOYE2: The SQL label for this file
does not match the label for the source $DATA17.PERSNL.EMPLOYEE.
```

If no mismatches occur, file recovery completes successfully. At the end of this recovery, verify the SQL objects recovered to a different location and perform Step 9 on page 11-26 if needed.

When recovering purged SQL objects to a different location, you do not need to re-create the source objects before recovery. Only the target objects must exist and match the source objects in terms of indexes, number of partitions, and so forth. You must alter the dumps of the purged objects to reset the INVALID and RELEASED flags before attempting recovery.

△ **Caution.** Use the TOFIRSTPURGE, TIME, or TOMATPOSITION option in the RECOVER FILES command to avoid replaying the purge operation on a target object. If you do not do this and the file-recovery process encounters a purge record for an object being recovered to a different location, the process terminates recovery of that object with these EMS messages:

```
NonStop TMF on \PLUTO *0437* RECOVER FILES [60]
FileRecovery Process #1: Encountered a purge record
for audited file $DATA16.PERSNL.EMPLIST while performing
FLABMOD REDO operation; Audit Trail Index #2, SNO #1,
RBA #22707360.
```

```
NonStop TMF on \PLUTO *0358* RECOVER FILES [60]
FileRecovery Process #1: Recovery on $DATA16.PERSNL.EMPLIST
has terminated.
```

You can recover SQL objects to a different location even when the source objects have not been purged. You can create the target objects to match the source, and then perform recovery to obtain a copy of your source objects. Transactions can be active against the source objects at the time of recovery.

△ **Caution.** If you use the MAP NAMES option of the RECOVER FILES command to recover files to a new location, you must immediately make new online dumps of the target data files recovered. Without these new dumps, you will not have file-recovery protection for those files, and subsequent file recovery operations can fail. In particular, if you later try to use old online dumps of the target files to recover the target files to a point beyond the time that the last RECOVER FILES command was issued, the file recovery process fails during the redo phase and transmits EMS message 175:

```
Encountered a File Hiatus record for audit file filename at
audit trail Index #index, SNO #sno, RBA #rba.
```

Other Recovery Methods

Other methods of recovering a dropped SQL table are possible, but they are riskier than the method just described. For example, you could use the same TMFCOM RECOVER FILES command to recover the SQL catalog as well as the table and its dependent objects. If successful, this method would eliminate the need to synchronize the timestamps in the catalog with those in the file labels, but note:

- You must recover the catalog precisely up to the time of the SQL table drop. Recovering to a later time causes you once again to lose the entries for the table.
- If you recover the catalog to the time of the table drop and if any updates were done to the catalog after the drop, the resulting catalog would miss those updates. Therefore, you should definitely avoid this method if it is possible that a catalog update occurred after the table drop. Such a step could create even more inconsistencies between the SQL catalog and the SQL objects. Consequently, you

can recover the table only by following all the steps under Recovery Example on page 11-20.

- You cannot use this method to recover a SQL object, along with its catalog, to a new location.

Recovering Catalogs

There are several ways you can recover a catalog that becomes corrupt.

Because the catalog tables are TMF audited tables, you can use the TMF file recovery method to recover the catalogs to a point where the catalogs were consistent. If any tables of the catalog have the undo-needed or redo-needed flag set, you should recover all the catalog tables by following the TMF recovery procedures for this method, described under File Recovery and File Recovery With the TIME Option on page 11-14.

△ **Caution.** Do not recover individual catalog tables.To keep an SQL catalog consistent, you must recover all the tables in the catalog as a set.

If TMF recovery fails or is not available, you might be able to correct the inconsistencies by using a licensed SQLCI2 process to change catalog entries. Inconsistencies can arise from the incorrect use of PUP commands (D-series only) and SCF commands (G-series only) or the incorrect use of licensed programs.

Note. A verify must be done on the restored table to find cases where USAGES entries are not updated for dependent objects that are registered in a catalog table different from the main table. If such cases are found, rows have to be added to the USAGES table of the other catalogs using a licensed SQLCI2.

Purging Damaged Objects With the CLEANUP Utility

The SQL/MP data dictionary, consisting of file labels and the catalog descriptions of the files, is extremely reliable because the TMF subsystem is used to audit the catalog tables and file labels. The catalog descriptions or file labels can become corrupt, however, through misuse of the BACKUP and RESTORE utilities, the TMF RECOVER FILES command, low-level system utilities such as PUP and TANDUMP, or because of software or hardware problems.

When the catalog description or file label for an object becomes corrupt, it might not be possible to purge the object by using the normal SQLCI PURGE or DROP command.

The CLEANUP utility, however, is specifically designed to purge a file, the file’s catalog description, and any dependent objects, when the SQL object is damaged.

△ **Caution.** The CLEANUP utility purges undamaged files in addition to damaged ones. The CLEANUP utility should never be used as a substitute for the SQL DROP statement or SQLCI PURGE command. The CLEANUP utility should be used only for removing objects that cannot be repaired using the TMF subsystem or removed by DROP or PURGE.

The command syntax for the CLEANUP utility is described in the *SQL/MP Reference Manual.*

When using the CLEANUP utility to remove damaged objects, follow these guidelines:

- Be careful when using a qualified file set or the ”!” format with the CLEANUP utility because you might inadvertently purge valid objects.
- To execute the CLEANUP utility, you must log on as the local super ID.
- The local super ID does not give you authority to purge objects on a remote node. Therefore, to purge objects distributed over multiple nodes, you must run the CLEANUP utility separately on each node.
- You cannot specify the CATALOGS option (for purging catalogs) and the SHADOWSONLY option (to enable or disable removal of shadow labels) in the same CLEANUP command.
- If the CLEANUP utility is used on a distributed database table, view, or index that has partitions or remote dependent objects, the remaining objects and the catalogs in which they are registered can still contain references to the objects purged with the CLEANUP utility. This situation is most likely in the case of a network-distributed object because the CLEANUP utility affects objects on the local node only. Be sure you remove the entire structure of a distributed object.
- The CLEANUP utility treats the dependents of an object as individual objects and purges them independently. Therefore, in unusual circumstances, it is possible to run the CLEANUP utility and still have dependent objects, partitions, views, or indexes that refer to a table that has been purged, or to be unable to apply the CLEANUP utility to objects because they are corrupt in an unusual way. These unusual circumstances are outlined under the discussion of the CLEANUP utility in the *SQL/MP Reference Manual*. In these cases, you must use a licensed SQLCI2 process to remove the offending catalog entries, and you must use the GOAWAY stand-alone utility to purge the disk file labels for the damaged objects.
- If an SQL program is dependent on an object being purged, the CLEANUP utility invalidates the program but does not purge it. If the program is stored in a Guardian file and is explicitly identified for deletion in a qualified file set, however, the CLEANUP utility purges the program. The CLEANUP utility does not purge SQL programs stored in OSS files.
- The CLEANUP utility does not transmit status information and operational results to the system log. Information about operational results is returned through SQLCI, however.

- You cannot specify the CLEANUP command within a user-defined TMF transaction. The CLEANUP utility protects the database, however, by automatically starting its own TMF transaction for each SQL object catalog description and file label operated upon. If the CLEANUP utility fails during execution, only the deletion of the last SQL object or partition is backed out.

Recovering From Peripheral Utility Program (PUP) Commands (D-series only)

SQL/MP introduces new relationships between volumes and nodes in a network. Disk names and node names are hard-coded references in the SQL/MP file labels and catalogs. Incorrect use of PUP commands or of the MAP NAMES option in RESTORE commands can lead to serious and possibly irreparable inconsistencies in an SQL/MP database.

If your site needs to use any of these PUP commands on a volume with SQL objects, you should carefully plan for a recovery method before using these commands: LABEL, RENAME, COPY, FORMAT, REMOVE, REVIVE, DOWN and UP. Each listed PUP command is discussed next.

PUP LABEL

PUP LABEL can irretrievably corrupt an SQL database. PUP LABEL should not be used on disks containing SQL catalogs or objects with a few exceptions.

PUP LABEL can be used on volumes that do not contain SQL catalogs or SQL objects. Do not attempt to recover a single volume when the database is distributed.

Use PUP LABEL on volumes with SQL objects in these situations:

- To label a disk that has been destroyed and has completely corrupt data
- To label all the disks on a node and recover the database with a complete restore
- To PUP LABEL a disk following repair or replacement if a volume with SQL objects has a catastrophic failure and no mirrored volume is available. In this situation, you should label the disk with its previous name.

If you use the PUP LABEL command, these are the steps for recovering the volume:

1. Use RESTORE to retrieve the nonaudited database residing on this volume; an example of the RESTORE command follows:

```
50>  RESTORE $TAPE, $VOL.*.*, AUTOCREATECATALOG ON,
              TAPEDATE, OPEN, LISTALL
```

2. Retrieve audited tables by using the most recent TMF online dumps and TMF file recovery. To initiate a file recovery of all files on $VOL, enter this command through one of the TMF interfaces; this example uses TMFCOM:

```
~  RECOVER FILES $VOL.*.*, CRASHOPEN OFF
```

3. Check that dependent objects residing on other volumes have also been recovered and re-create objects as necessary.

   For example, a table resides on another volume, but a dependent index resides on the newly labeled disk volume. If the index was not recovered, re-create the index. You should also check that all view definitions are current and that all shorthand views were recovered. Re-create any views that were not recovered.

△ **Caution.** Do not use RESTORE to restore an index; doing so might cause inconsistencies in the database.

---

4. Depending on the date and time of the most recent BACKUP and TMF recovery point, the restore operations might not be able to retrieve a consistent database with mixed audited and nonaudited files. Manually resolve any inconsistencies between audited and nonaudited database files.
5. SQL compile any programs that were invalidated by this process and that reside on other volumes. Also, SQL compile all programs restored to $VOL to validate them and register them in a catalog.
6. Verify the database by using the VERIFY utility; an example of the VERIFY command follows:

```
>>  VERIFY $VOL.*.*;
```

7. Drop and re-create any shorthand views that might have been left in an invalid state. Any invalid shorthand views will be identified in Step 6.
8. Make new TMF online dumps of all catalogs and audited objects on the volume.

PUP RENAME

△ **Caution.** The use of PUP RENAME is extremely dangerous because it can corrupt a database. The PUP RENAME operation renames the files on a volume, but SQL catalogs and file labels still contain the old name. Do not use PUP RENAME on disks that contain SQL catalogs or objects. PUP RENAME should only be used for volumes that do not contain SQL catalogs or other SQL objects.

If a volume is renamed inadvertently, use the PUP RENAME command to rename the volume to its previous name.

If a volume must be renamed, only a knowledgeable database administrator should attempt the operation. Files must be backed up to tape and then restored to the renamed volume. This task is similar to that described in Steps for Moving a Database on page 9-25. Note that volumes can be recovered only if all the objects on the renamed volume are described in catalogs on the same volume. If this is the case, use the Guardian BACKUP and RESTORE utilities to back up the volume, as follows:

1. Determine all the SQL objects to be renamed and all dependencies. Produce hard-copy reports containing this information.

2. Create an EDIT file containing CREATE CATALOG and ALTER TABLE statements to re-create the catalogs and reset the security of the catalog tables.
3. Back up the volume by using a file-mode BACKUP command.

Next, use PUP RENAME to rename the disk. You might first want to label the disk with PUP LABEL to clear all the old files.

Finally, restore the files as follows:

1. Re-create the catalogs on the renamed volume using the file created in Step 2 as the input file for the SQLCI OBEY command.
2. Restore the volume, mapping the old volume names to the new volume names in the RESTORE command. Map the objects to the new catalogs.
3. Verify the database by using the VERIFY utility; an example of the VERIFY command follows:

```
>>  VERIFY $VOL.*.*;
```

4. SQL compile all the programs with new DEFINEs to revalidate the programs.

PUP FORMAT

The PUP FORMAT command erases all the information on a disk volume. If a disk volume is formatted, you must follow the same procedure as you would with the PUP LABEL command.

PUP REMOVE and PUP REVIVE

You can use the PUP REMOVE command on a mirrored volume pair to make one half of the pair inactive. The active disk drive of the mirrored pair continues to maintain the current database, without the protection of mirroring.

After the disk drive is removed, you can bring the disk up as a phantom drive (without a name), label the disk with another volume name, or reuse the disk in any other way. This operation is often done on nodes where nonmirrored disk space is needed for a short time.

Later, you can return the previously removed disk drive to its original mirrored state by performing a PUP REVIVE.

You should not use the removed drive to store production SQL database files. You typically use the drive for a test database or for temporary space for sort files. The use of the drive must ensure that you can make the volume inactive and revive the drive back to its original mirrored configuration with no effect on the original database.

PUP DOWN (or PUP REMOVE) and PUP UP

Use the PUP DOWN or PUP REMOVE command on a volume to put a particular volume out of operation; you can later use the PUP UP command to put the same disk back into operation. There is no danger of inconsistency as long as the disk brought up is identical to the disk brought down. You should always perform a PUP STOPOPENS on the volume and a PUP REFRESH on the volume to ensure valid file labels before you make the volume inactive.

△ **Caution.** You cannot use PUP DOWN or PUP REMOVE on a volume and replace the volume with an older version of that same volume without causing inconsistencies in the database.

The only exception to the preceding rule is if the entire database has been consistently brought down as a unit. For example, suppose that you use PUP DOWN to bring down all the backup volumes of the mirrored pairs containing SQL objects in a consistent state. The other mirrored set continues the active database, but the inactive mirrors also contain a set of consistent SQL objects.

You can also use PUP DOWN bring down the active database and PUP UP to bring up the saved database in a database swapping technique. This technique might be useful for testing scenarios. As long as you bring each set of mirrors down and then up together, each copy of the database continues to be consistent.

Note. The corresponding SCF commands (G-series only) for the PUP commands (D-series only) are listed on page 11-35. Use the same strategies for recovering from SCF commands as indicated for PUP commands.

SCF Commands (G-series only)

In G-series RVUs, PUP functions are performed by SCF.

SCF is an interactive interface for configuring, controlling, and collecting information from a subsystem and its objects. SCF enables you to configure and reconfigure devices, processes, and some system variables while your NonStop S-series server is online.

SCF ALTER DISK, LABEL

The ALTER DISK, LABEL command erases the existing files and writes a volume label on a new or previously labeled volume.

SCF RENAME

The RENAME command replaces only the default or alternate name of the volume (VOLNAME or ALTNAME options). If you use the ALTER DISK, LABEL command to change both names, all files on the volume are deleted.

SCF INITIALIZE DISK

The INITIALIZE DISK command erases existing files, labels the disk, and starts it.

SCF STOP DISK and SCF START DISK

The SCF STOP DISK command performs an implicit remove. (An implicit remove is also performed when the system is shut down.)

The SCF START DISK command performs an implicit revive, if needed, to update one half of a mirrored volume.

SCF STOP and SCF START

The SCF STOP command stops the object in an orderly way. The device is not stopped until the current activity ends.

The SCF START command starts the object or process if it is in a STOPPED state, making it available to user processes. START DISK is also used to revive one half of a mirrored volume that is in a STOPPED state, substate DOWN.

Managing Shadow Disk Labels

Shadow labels are the internal labels created by the disk process when SQL objects are dropped within a transaction. Normally, these labels are deleted soon after the transaction completes. In some situations, however, especially during abnormal processing or a system crash, these labels are not deleted until file recovery is performed.

Usually, shadow labels do not cause any problems on the system, but if users issue subsequent CREATE statements to create objects with the same file name as the shadowed label, the create operation fails.

You can remove shadow labels with the SHADOWSONLY option of the PURGE utility.

Identifying Shadow Labels

You might see shadow labels for a short period of time following the DROP or PURGE command; this is normal. Sometimes shadow labels are left on the system.

You can detect shadow labels by using either the DSAP or FILEINFO commands. You must run the command on all the volumes or qualified file-set lists that might have resident SQL labels to check that no shadow labels exist.

DSAP reports the message “(SQL Shadow)” after the file name to indicate the file label is shadow only. DSAP might also indicate that the file has doubly allocated extents. These shadow labels might be the result of active DROP or PURGE commands that have not removed the shadow label. The extents are allocated until the shadow label is dropped or removed.

This command illustrates using DSAP on volume $VOL1:

```
51>  DSAP $VOL1 DETAIL SQL
```

FILEINFO indicates shadow labels by displaying an S in the file type field of the report. If you request a detailed FILEINFO display, SQL SHADOW LABEL appears in the line describing file type; however, some other information is not available, such as key information and index information.

These commands use FILEINFO to detect shadow labels:

```
>> FILEINFO *.*.*, SHADOWS
```

```
>> FILEINFO *.*.*, DETAIL SHADOWS
```

Removing Shadow Labels

You can remove shadow labels from the system with either the PURGE or CLEANUP utility. The SHADOWSONLY option of these utilities enables or disables the purging of shadow labels.

The SHADOWSONLY option is like a toggle. If you specify SHADOWSONLY, only shadow labels in the file set list are purged; other files in the file set list are not considered for the purge. If you omit the SHADOWSONLY option, no shadow labels are affected; only files in the file set list are purged.

When you use the SHADOWSONLY option, follow these guidelines.

- You must be logged on as the super ID. If you are not the super ID user, a warning is issued and nothing is purged.
- When you run the PURGE or CLEANUP command to remove shadow labels, no user-defined TMF transaction should be active. These utility commands are meant to purge the shadow labels produced because of a damaged system or hardware malfunction, but not to purge the ones produced in normal operation.

This command purges shadow labels by using PURGE:

```
>> PURGE  $VOL1.*.*, SHADOWSONLY;
```

This command purges shadow labels by using the CLEANUP utility:

```
>> CLEANUP  $VOL1.*.*, SHADOWSONLY;
```

12 Managing a Distributed Database

Databases can be distributed over disk volumes on a single system (node) or in a network of nodes. Likewise, application programs can be distributed across processors in a single node or in a network.

When managing a database distributed across volumes or nodes, use the same SQL statements you would use with a nondistributed database. When accessing a distributed SQL object, some SQL statements enable you to use distinct file names that refer to individual partitions of the object. For other statements, however, a partition name refers to the entire object rather than to the individual partition.

The distribution issues discussed in this section are divided into the general areas of locally distributed databases (distributed over two or more disk volumes on the same node) and network-distributed databases.

Managing a Locally Distributed Database

An SQL/MP database is locally distributed if any tables, views, or indexes are partitioned over two or more volumes. The goals for managing a locally distributed database are:

- Using the total available processing power of the system while balancing the workload
- Enabling very large data files to physically spread across multiple disk volumes while accessed as single files

Using DEFINEs for Logical Name Mapping

When you are working with distributed objects, you should always fully qualify each reference, either in each statement or by using DEFINEs.

Use DEFINEs for a distributed database in the same way you would for a nondistributed database. You might want to create DEFINE names for each partition of the object because the partitions might be accessed separately. For a distributed object, you can include the partition number in the DEFINE name to avoid any confusion about the applicable partition in this format:

```
=partition-number_define-name
```

These DEFINE names are examples of distributed names:

```
=PART1_EMPLOYEE, CLASS MAP, FILE \LOCAL1.$VOL1.PERSNL.EMPLOYEE
=PART2_EMPLOYEE, CLASS MAP, FILE \LOCAL1.$VOL2.PERSNL.EMPLOYEE
```

Maintaining Local Autonomy

Local autonomy implies that a DML request, initiated either interactively or with an application program, can access local data, regardless of the availability of remote dependent objects or other local dependent objects if the local data can satisfy the request.

For example, if a table named PARTS is partitioned with a partition on $VOL1 and another partition on $VOL2, a query of PARTS can access the partition on $VOL2 regardless of the availability of $VOL1. The $VOL2 partition can be opened upon demand for its access if the CONTROL TABLE OPEN ACCESSED statement is in effect.

Similarly, if a query tries to access the table named EMPLOYEE, residing on $VOL2, through an index named IEMP2, residing on $VOL1, the query can be completed regardless of the availability of $VOL1. If $VOL1 is not available, SQL/MP automatically tries to find an alternate path. For more information about access paths, see Maintaining Local Autonomy in a Network on page 12-5.

Managing a Network-Distributed Database

NonStop systems can be linked together by communication lines to create a network. Each system on the network is called a node.

An SQL/MP database or application can be distributed over a network of nodes in several ways:

- Tables or indexes are partitioned over two or more nodes.
- Tables, indexes, views, or programs reside on two or more nodes.
- A local shorthand view or index references a remote base table.
- Local programs access remote tables or views or use a remote access path.
- Remote programs access local tables, views, or indexes.

The goals for managing a network-distributed database area follows:

- Efficiently share data among users located remotely from one another.
- Allow for local identity and control while sharing information.
- Eliminate duplication of data.
- Increase the local computing power by the aggregate total of the computing power of the network.

Naming Nodes

An SQL/MP system requires a node name. An SQL/MP system in a network requires a node number in addition to a node name.

After an SQL/MP database is created using a node name and node number, you should minimize changes to the name and number. The node name is expanded in the catalog entries, and the node number is entered in the file labels throughout the database. Choose your node name and number carefully so that you will not need to change them in the future.

Using DEFINEs for Network Object Names

When you work with distributed objects, always fully qualify each reference, either in each statement or by using DEFINEs.

Use DEFINEs for a network distributed database in the same way you would for a local system. For a network distributed system, however, you should include the node name in the DEFINE name to avoid any confusion about the applicable node.

A possible format for DEFINE names that include the node name follows:

```
=node-name_define-name
```

These DEFINE names are examples of distributed names:

```
=REMOTE1_EMPLOYEE, CLASS MAP, FILE
\REMOTE1.$VOL1.PERSNL.EMPLOYEE
=LOCAL_XEMP, CLASS MAP, FILE \LOCAL.$VOL3.PERSNL.EMPLOYEE
```

Always qualify the =_DEFAULTS DEFINE with the fully qualified name, including the node name. Likewise, you should always fully qualify names in VOLUME and CATALOG commands.

When using the SYSTEM command with SQLCI, the node (system) you specify is stored in the *volume.subvolume* string. Then, when you specify Guardian names without fully qualifying them, these names are expanded with the fully qualified volume string that includes the node you specified.

The CATALOG string also stores a fully qualified catalog name, such as \SYS1.$VOL1.SALES. The CATALOG command qualifies a partially specified catalog name by using the current node and does *not* automatically expand the name by using the node name stored in the SYSTEM *volume.subvolume* string.

Using Catalogs in a Network

Each node must have a system catalog and catalogs for the objects located on that node. A catalog can hold the descriptions of objects that reside only on the same node. For example, you cannot describe a table on node \SYSA in a catalog that resides on node \SYSB.

For distributed and partitioned tables or indexes, you must define a catalog to describe the partition resident on that node. Partitioned tables and indexes must have the table or index description in a catalog on each node where any partition resides; consequently, each node with a partition maintains a copy of the description.

When a distributed SQL object is created, the fully qualified Guardian name of the object (\system.\$*vol*.*subvol*.*filename*) is coded in each catalog that contains a description of the object and also in the file label.

Managing Network Security

Managing a network-distributed database has additional demands on security and authorization schemes.

All users of a distributed node must have remote passwords for remote access. All remote objects and local objects must be secured for network access.

In addition to the authority and security for the SQL objects, statements that require access to catalogs also require that the remote catalogs be secured for network access.

For security in a local node in a network, the rule for authority is this: to perform DDL operations on existing objects you must be the local owner of an object, a remote owner with authority to purge the object, or the super ID.

Authority to purge the object is required to drop a table, program, or view. Authority to purge the underlying table is required to drop an index or constraint.

For security on a remote node, the rule for authority is this: to have the authority to perform DDL operations on an existing object, you must be the remote owner of the object with authority to purge the object. To drop a table, program, or view, you must have authority to purge that object. To drop an index or constraint, you must have authority to purge the underlying table. The super ID does not have the remote capabilities that the super ID has in the local environment.

Group managers, like other users, must meet the normal purge authority requirements to perform DDL operations on a remote object; however, a group manager can read, write, or execute any object owned by any member of the group. The remote object must be secured for remote access with the letters U, C, or N.

Maintaining Local Autonomy in a Network

In the context of a network distributed database, local autonomy ensures that a user can access local data regardless of the availability of remote dependent objects. For example, if a table is partitioned with a portion on \SYS1 and another portion on \SYS2, a local user of \SYS1 can access the local partition of the table when \SYS2 is not available. This access is useful, of course, only if the needed rows reside in the partition on \SYS1.

Each partition of a distributed table or index is described in a catalog on each local node. This duplication of description allows for local autonomy. Access to the primary partition of a distributed table or index is not required to access any other partition. For index-only scans, however, the partition of the base table that corresponds to the requested data range must be accessible.

In a distributed application, you can maximize local autonomy by referring to a local partition as a table name in local programs.

When the program refers to a local partition, the SQL compiler checks for information about the table in a local catalog. When a program refers to a remote partition, the SQL compiler must check for information about the table in the remote catalog. If the remote node is down, SQL compilation fails. When the local node is up and the data is available locally, the local SQL compilation can succeed.

By using DEFINEs and a program to refer to tables, and by associating the DEFINEs with local partitions, you increase the possibilities for successful SQL compilations for programs that use distributed data. This advantage applies to explicit SQL compilations, automatic SQL recompilations, and dynamic SQL statement compilations.

Local Autonomy and DML Operations

Local autonomy applies to run time DML access. When a node required for an access path is detected as unavailable at run time, the DML statement can be SQL recompiled to find another access path using the available nodes. If there is another access path, the statement is executed.

An INSERT, UPDATE, or DELETE statement cannot complete if the statement tries to write or delete a row in an unavailable table, index, or partition of a table or index.

For example, a table and two indexes are located on two different nodes:

- Table X (Columns A1, B1, C1, C2, C3, C4) resides on \NODE1
- Index A, using Columns A1 and B1, resides on \NODE3
- Index B, using Columns A1 and C4, resides on \NODE1

Using this table and these indexes, these scenarios illustrate some of the features and restrictions of local autonomy for DML operations:

- If a query uses Index A as the access path but \NODE3 is down, the query is recompiled to attempt to access the data by using Index B or by using the primary key residing on \NODE1.
- If an INSERT statement tries to insert a row into Table X with values for all the columns, the insert fails if \NODE3 is down, because Index A cannot be updated.
- If an UPDATE statement tries to update Columns C1 and C2 of Table X, the update completes although \NODE3 is unavailable, because Index A on \NODE3 is not required for the update.

Local Autonomy and DDL Operations

Local autonomy does not usually apply to DDL operations, which usually require the availability of all dependent objects affected by the operation. For example, the CREATE INDEX statement requires that all partitions and protection views of an underlying table be available. Certain DDL statements, however, such as ALTER TABLE PARTONLY, can be performed successfully on the partition involved.

Local Autonomy and SQL Compilations

Autonomy is also not completely supported at SQL compile time. A program is compiled with the best query execution plan only if all the local and remote catalogs of the tables, views, and associated indexes are available. If all the required information is available and the compilation is successful, the program is entered in the PROGRAMS and USAGES tables and marked valid in the catalog and in the program file label.

SQL compilations that occur when nodes are unavailable can still register the compiled programs in the PROGRAMS and USAGES catalog tables and mark the programs as valid if the FORCE option was chosen. Those statements that could not be compiled with the best query execution plan are marked invalid on a statement-by-statement basis. The invalid statements will be automatically recompiled at run time.

You should SQL compile programs used in a network only when all referenced nodes are available so that the compiler can create the best query execution plan. You should also use the RECOMPILE option of the SQL compiler so that the automatic recompilation can occur at run time if access paths become unavailable.

For more information about the access strategies of programs, see Section 10, Managing Database Applications.

Network Availability and Use

Network lines have a direct impact on the performance of a distributed database. The line speed, network routing, and network message traffic use significantly affect response time. The setup and management of a network should be thoroughly studied by the system or network manager. For more information, see the *Communications Management Interface (CMI) Operator Reference Manual*.

Remote Node Availability

Unavailable remote nodes can prevent programs that require data from those nodes from obtaining needed data. You can, however, distribute data in an SQL/MP database so that local data is stored locally and is available locally regardless of remote node availability.

To ensure automatic recompilation for programs when access paths become unavailable, you should explicitly SQL compile programs in a distributed environment with the RECOMPILE compiler option.

Automatic recompilation can decrease performance for the amount of time required to recompile the program or statement for the first time. When the same program is again executed after the node has been restored to the network, the program is not automatically recompiled but uses the original plan determined by the SQL compiler when the program was explicitly compiled. A running process, however, does not revert to the original query execution plan; only a newly started process would attempt to use that plan.

Creating a Distributed Database

Objects can be distributed individually or distributed as partitions of tables or indexes.

Objects are distributed at creation by fully qualifying the names in the CREATE statement or DEFINE. If you have the authority, you can create objects on a remote node or create local objects that refer to remote objects.

All nodes referred to in a CREATE statement must be available to create an object.

This example creates a local shorthand view on both a local and a remote table:

```
>> CREATE VIEW \LOCAL.$VOL1.SALES.REPORDS
+>    AS SELECT A.SALESREP, A.ORDERNUM, A.DELIV_DATE,
+>       B.CUSTNUM, B.CUSTNAME
+>    FROM \LOCAL.$VOL1.SALES.ORDERS A,
+>         \REMOTE.$VOL4.SALES.CUSTOMER B
+>    WHERE A.CUSTNUM = B.CUSTNUM
+>    CATALOG \LOCAL.$VOL1.SALES;
--- SQL operation complete.
```

This example creates a remote table and a local index on the table. The table and index are registered in catalogs on the nodes on which they reside.

```
>> CREATE TABLE \REMOTE.$VOL4.SALES.PARTS
+>   (PARTNUM       NUMERIC (4) UNSIGNED NO DEFAULT     NOT NULL,
+>    PARTDESC      CHARACTER (18)       NO DEFAULT     NOT NULL,
+>    PRICE         NUMERIC (8,2)        NO DEFAULT     NOT NULL,
+>    QTY_AVAILABLE NUMERIC (7)          DEFAULT SYSTEM NOT NULL,
+>    PRIMARY KEY   PARTNUM)
+>  CATALOG \REMOTE.$VOL4.SALES
+>  SECURE "NNOC";
--- SQL operation complete.
>> CREATE INDEX \LOCAL.$VOL1.SALES.XPARTDES
+>   ON \REMOTE.$VOL4.SALES.PARTS (PARTDESC)
+>   CATALOG \LOCAL.$VOL1.SALES;
--- SQL operation complete.
```

This example creates a partitioned table with partitions on both a local node and a remote node:

```
>>  CREATE TABLE  \LOCAL.$VOL1.INVENT.PARTLOC
+>   (LOC_CODE     CHARACTER (3)           NO DEFAULT  NOT NULL,
+>    PARTNUM      NUMERIC (4) UNSIGNED    NO DEFAULT  NOT NULL,
+>    QTY_ON_HAND  NUMERIC (7)             NO DEFAULT  NOT NULL,
+>      PRIMARY KEY (LOC_CODE, PARTNUM ))
+>      CATALOG \LOCAL.$VOL1.INVENT
+>      ORGANIZATION KEY SEQUENCED
+>      PARTITION (\REMOTE1.$VOL2.INVENT.PARTLOC
+>                    CATALOG \REMOTE1.$VOL2.INVENT
+>                    FIRST KEY "G00",
+>                 \REMOTE2.$VOL3.INVENT.PARTLOC
+>                    CATALOG \REMOTE2.$VOL3.INVENT
+>                    FIRST KEY "P00")
+>      SECURE "NNOO";
--- SQL operation complete.
```

When creating a table on a remote system, the local system default multibyte character set is used.

Altering Distributed Objects

You can perform alter operations on distributed databases, as described under Altering Database Objects on page 7-13.

The ALTER statement allows these operations to be performed on partitions independently of the other partitions: you can allocate or deallocate extents or specify a different MAXEXTENTS value for each partition. To alter these attributes, use the PARTONLY option of the ALTER statement. PARTONLY applies to tables or indexes. Alterations that do not allow the PARTONLY option affect the entire table or index (all distributed partitions of a table or index).

This example demonstrates altering the maximum extents. In the example, $VOL1.SALES.ORDERS is a secondary partition of a partitioned table.

```
>>  ALTER TABLE $VOL1.SALES.ORDERS PARTONLY MAXEXTENTS 124;
--- SQL operation complete.
```

You can also add, split, or drop partitions of tables or indexes, as explained in Section 7, Adding, Altering, Removing, and Renaming Database Objects.

Note that ALTER operations are subject to versioning requirements. For example, you cannot use the WITH SHARED ACCESS option with a split, merge, or move boundary request unless each source object and each target object reside on a node running version 315 or later of SQL/MP software. You can only perform DML or DDL operations on tables with extended partition arrays from nodes running version 320 or later of SQL/MP software.

Dropping Distributed Objects

When you DROP or PURGE a distributed table, all indexes, partitions, and views must be accessible, in addition to the catalogs that describe these objects. If you do not have the authority to drop a shorthand view, the operation only invalidates the view.

When you specify dropping any partition of a table or index, the operation drops the entire table or index. You cannot specify a DROP, PURGE, or PURGEDATA operation on any individual partition.

You can also drop empty partitions of a table or index with the ALTER TABLE or ALTER INDEX statement. Dropping partitions is discussed in Dropping Partitions of Tables and Indexes on page 7-32.

Enhancing Performance for a Distributed Database

The performance issues of a distributed database encompass those of a local database and also include these:

- Effective use of local partitions or indexes
- Use of replicated data to increase local performance
- Use of remote servers to increase performance

These issues, not covered in this manual, also affect performance of a distributed database:

- Network availability and use
- Remote node availability

For additional information about enhancing performance, see Section 14, Enhancing Performance.

Using Local Partitions and Indexes

Defining local partitions of a table, so that the local partition can satisfy a significant number of local queries, can improve performance. Also, the local partition remains available to satisfy the queries even when other nodes are unavailable.

Performance might also improve for queries on a remote table if a local index exists to resolve queries locally. If the local index columns can resolve a query, the SQL executor does not need to query the remote table.

Before defining the index or local partitions, you must weigh the benefits against any performance considerations that occur when the underlying table is modified. Insert, update, or delete operations on the table from anywhere in the network also require access to the local index and possibly to the local partition.

This example creates a local index on a remote table:

```
>> CREATE TABLE \REMOTE.$VOL1.SALES.CUSTOMER (
+>   (CUSTNUM    NUMERIC (4) UNSIGNED   NO DEFAULT      NOT NULL,
+>    CUSTNAME   CHARACTER (18)         NO DEFAULT      NOT NULL,
+>    STREET     CHARACTER (22)         NO DEFAULT      NOT NULL,
+>    CITY       CHARACTER (14)         NO DEFAULT      NOT NULL,
+>    STATE      CHARACTER (12)         DEFAULT SYSTEM  NOT NULL,
+>    POSTCODE   CHARACTER (10)         NO DEFAULT      NOT NULL,
+>    CREDIT     CHARACTER (2)          DEFAULT "C1"    NOT NULL,
+>    PRIMARY KEY  CUSTNUM)
+>  CATALOG SALES
+>  ORGANIZATION KEY SEQUENCED
--- SQL operation complete.

>>  CREATE INDEX \LOCAL.$DATA1.SALES.XCUSTNAM
+>    ON SALES.CUSTOMER (CUSTNAME)
+>    CATALOG \LOCAL.$DATA1.SALES;
--- SQL operation complete.
```

This query can be satisfied by information in the local index. The query should be able to be completed without retrieving data from the remote underlying table.

```
>> SELECT CUSTNAME
+> FROM  \REMOTE.$VOL1.SALES.XCUSTNAM;
```

Supporting Replicated Data Through Indexes

SQL/MP does not specifically support replicated data except through indexes. You can create an index on a remote node with all the columns in the table as keys in the index, provided the index row length does not exceed the maximum length. This technique effectively provides system support for replication.

SQL/MP supports replication through indexes as follows:

- First, create an index on a remote node that specifies all the columns except the primary key columns in a local table; the primary key columns are included in the index automatically.
- When you update the local table, SQL/MP automatically updates the index at the remote node. Because the index contains all columns in the table, this approach is just the same as updating a replica of the table at the remote node.
- When you run a query at the remote node to select data from the table, SQL/MP selects the data from the index because the index is local to the query.
- When you run a query at the local node to select data from the table, SQL/MP selects the data from the table because the table is local to the query.

Using Remote Servers

When you use a network-distributed database, you can often control whether remote data is updated directly by a local server or indirectly by a remote server. Any local program can update or retrieve data directly by using the remote I/O capabilities of the Guardian file system and disk process.

Alternatively, when you need to update data stored at a remote node, you can send a message containing an update request to a server at that remote node. Ultimately, this issue might be one of performance and processing power distribution across nodes.

One of the main advantages of using a remote server for distributed processing is to reduce the amount of data sent across communication lines. One message makes the request of the server at the remote node. Then, that server manages all access to, and updating of, the remote data. This approach reduces message traffic on the slower communication lines and increases performance.

Managing Processor Usage in a Distributed Environment

For a query that executes in parallel in a distributed system or network, you can choose a specified set of processors in which the query will run. The remaining processors are free for other tasks—for example, executing a different type of query for another application.

You control processor usage by using the _SQL_CMP_CPUS DEFINE. Before compiling a query, you add this DEFINE to select a set of “usable” processors. The optimizer chooses an access plan that uses only the allowable processors for executor server processes (ESPs) and for temporary files chosen for repartitioning.

The DEFINE influences all parallel execution plans compiled while the DEFINE is in effect. You can reset the DEFINE to change the usable processors before compiling other queries, so different queries can have different “usable” processors.

By determining that certain sets of queries (or applications) run on certain processors, you can improve the performance and manageability of a distributed system or network—especially one used for multiple purposes. You can select usable processors in a single node or across multiple nodes.

Design Examples

These examples suggest the potential benefits of using the _SQL_CMP_CPUS DEFINE:

- A development-and-test environment and a production environment share a single system. By limiting the development and test activity to certain processors, you enhance the performance of the production queries.
- You have an existing OLTP environment, possibly supported by batch applications, and an expanding DSS environment. As (or before) you scale up the DSS environment, it shares processor resources with the OLTP environment. By segregating the batch queries and DSS queries into separate processors, you can improve the performance of both.

  The mixed workload feature already allows you to prioritize DP2 requests for different queries. Limiting processor use for different queries increases your ability to manage query performance in a multiple-use environment. For a brief description of the mixed workload feature, see the *Introduction to NonStop SQL/MP*.

These considerations apply to the _SQL_CMP_CPUS DEFINE:

- It only operates on queries that use parallel execution plans. The DEFINE influences the location of ESPs but not the location of the master executor, so serial plans are not affected by it.
- It does not determine the processor locations of multiple sort processes used in a parallel execution plan. However, the ESPs communicating with the sort processes are limited to the processors specified in the DEFINE.
- It does not determine the processor locations of disk processes involved in the query. Partitions accessed by the query determine the locations. Each partition is accessed by a disk process in the processor managing that partition’s disk volume, regardless of whether the processor is specified as usable by the DEFINE.

Consequently, when you use this DEFINE to limit the usable processors, you can still read tables partitioned across disk volumes primaried to “unusable” processors. This gives you two basic design options:

- You can limit the usable processors to a smaller or different set than the set of processors that manages access to the database. This approach makes a single database available to different types of queries (or applications), but you should limit the processors used in each type of query.
- You can partition your tables so that the same set of processors manages data access and is specified as usable for parallel queries. This choice lets you completely segregate the processors for separate uses.

You can use this feature together with the SMF product to manage resources in a distributed environment. For example, to set up a system in which a specified subset of processors performs both parallel query operations and data access for a particular application:

1. Using SMF, create a storage pool and assign it physical volumes that are all primaried to a specified set of processors.
2. Create partitioned tables and indexes for the application; use virtual volumes associated with the storage pool defined in Step 1.
3. Set the _SQL_CMP_CPUS DEFINE to limit parallel queries (ESPs) to the specified set of processors.
4. Compile the programs for the application.

These steps suggest in a general way how to use these two features together. Specific uses will vary according to the requirements of your environment.

For an overview of the benefits of using SMF to manage disk volumes for partitioned SQL tables, see Creating Partitions on a System That Uses SMF on page 5-32. For more information about using SMF, see the *Storage Management Foundation User's Guide*.

SQL Compilation and the CPU Usage DEFINE

The _SQL_CMP_CPUS DEFINE affects the compilation of queries. In a parallel execution plan, the compiler assigns ESPs to the processors specified as usable by the DEFINE. If a processor is unavailable at compilation time, the compiler does not assign any ESPs to that processor, although the DEFINE has specified it as usable.

If a processor that was specified as usable is unavailable at run time, the executor reassigns its ESP to another processor. The substitute processor does not have to be described as usable by the DEFINE.

You can reset the DEFINE after you compile (and run) a query. If you need to recompile the query, or if it undergoes automatic recompilation, the compiler will use the potentially different DEFINE values in effect at the time of recompilation. If you want the compiler to use the original DEFINE values, use the STOREDDEFINES option when you first compile the query. For more information about the STOREDDEFINES option, see Using DEFINEs During Compilation on page 10-32.

Using the Processor Usage DEFINE

You specify which processors are usable in a given system by using the FILE clause of the _SQL_CMP_CPUS DEFINE. The FILE clause has the format X*hhhh*. The four variables (*hhhh*) are hex characters that identify up to 16 processors in a system. Each hex character identifies four processors, as follows:

```
First hex character:   CPUs 0-3
Second hex character:  CPUs 4-7
```

```
Third hex character:   CPUs 8-11
Fourth hex character:  CPUs 12-15
```

SQL translates each hex character into its binary counterpart. Each bit represents one processor. If a bit is on, SQL uses the corresponding processor. If a bit is off, SQL does not use the corresponding processor.

This hex conversion table shows how each hex character represents four processors with on or off bits:

Hex	**Binary (CPUs on or off)**	**Hex**	**Binary (CPUs on or off)**
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

If you have fewer than 16 processors in a system, you can use fewer than four hex characters in the FILE clause. SQL assumes that missing trailing hex characters are zero, so the processors they represent are not used.

For a complete description of the syntax of the _SQL_CMP_CPUS DEFINE, see the *SQL/MP Reference Manual*.

To use hex conversion to specify the processors you want to use:

1. Determine the processor numbers you want to use. For example, suppose that you want to use processors 0, 1, 2, 3, 8, 10, 12, and 15.
2. Lay out the processor numbers from 0 through 15 (or your highest processor number). Place a 1 under each specified processor you want to use. Place a 0 (zero) under each specified processor that should not be used.
3. Convert the numbers to their hex counterparts.
4. Use the hex characters in the FILE clause of the _SQL_CMP_CPUS DEFINE.

In this example, suppose that you want to use processors 0, 1, 2, 3, 8, 10, 12, and 15:

```
CPU #:     + 0 1 2 3 + 4 5 6 7 + 8 9 10 11 + 12 13 14 15
ON/OFF:    + 1 1 1 1 + 0 0 0 0 + 1 0  1  0 +  1  0  0  1
HEX value:      F         0          A            9
```

Using this conversion, you can set the DEFINE as follows:

```
>> ADD DEFINE =_SQL_CMP_CPUS_SYS1, CLASS MAP, FILE XF0A9;
```

In the preceding example, the SYS1 specifies the system for which you are controlling processor usage. If you do not include a system name, the DEFINE applies to the current system.

This example limits the processor usage on \MYSYS to processors 0 and 1:

```
>> ADD DEFINE =_SQL_CMP_CPUS_MYSYS, CLASS MAP, FILE XC;
```

This example limits the processor usage on \DEV1 to processors 2 and 3:

```
>> ADD DEFINE =_SQL_CMP_CPUS_DEV1, CLASS MAP, FILE X3;
```

This example limits the processor usage on \PROD1 to processors 0, 2, 4, 6, 8, 10, 12, and 14:

```
>> ADD DEFINE =_SQL_CMP_CPUS_PROD1, CLASS MAP, FILE XAAAA;
```

Changing Network Environments

Network environments are often subject to change. Nodes can be added or removed, system configurations at each node can change, the operating system can be updated independently at each node, communication line speeds or types can change, and the needs of the node with respect to the database or application can change.

Many of these changes do not affect the SQL database or environment and need not concern you if you are a system manager. Certain changes, however, can cause problems or affect the SQL environment and must be anticipated.

Generally, these situations need to be considered on a case-by-case basis:

- A new node is added to the network. This addition does not affect the existing database scheme. To access this node and incorporate it into the overall environment, however, network passwords and security must be added to all other nodes. After SQL is initialized on this node, SQL objects can be placed on the new node.
- An existing node is permanently removed from the network. All SQL objects that refer to this node and all distributed SQL objects using this node must be purged before the node is removed. If objects referring to this node are left in the environment, these objects will receive errors from SQL on the remaining nodes.
- A node must be renamed or given a new node number. This procedure can be complex because the objects throughout the network that refer to an object, partition, view, or index on this node have the node name and number embedded in the file labels and referred to in the catalogs. For more information, see Renaming or Renumbering a Node on page 9-32.
- The operating system at a node is updated. Usually nodes run compatible but different operating systems. Consult the current software release documents for compatibility issues between operating system releases.
- Communication to a node is lost. Situations can occur where nodes become unavailable for various reasons. If at all possible, network transactions should be

quiesced before the communication loss for planned downtime. Transactions on other nodes will continue, and might not be adversely affected, as a result of local autonomy, but you might need to use one of the TMF interfaces (such as TMFCOM) to back out or commit the transactions manually. Transactions requiring data on the unavailable node return errors. After communication is restored, transactions can proceed normally.

- Recovery takes place for a system crash on a single node. If a node crashes, you can recover it by using a TMF recovery method. HP recommends that you initiate the START TMF, TRANSACTIONS OFF operation at the crashed node. This approach enables the TMF subsystem to resolve any network-distributed transactions active at the time of the crash and to attempt volume recovery.

  Keeping TRANSACTIONS OFF during this procedure enables the function to complete successfully, before new transactions are introduced to the database. If the node is successfully recovered, transactions can then be turned on. For additional information, see the *TMF Operations and Recovery Guide.*

- A number of situations can cause severe problems with the consistency of a SQL/MP database. Various techniques can resolve these problems. You should not, however, attempt these operations without help from your service provider:
  - Recovering a node with distributed objects by using the TIME option of the TMF subsystem, which can cause an inconsistent database
  - Performing a RESTORE of objects on a node, such that the restored objects are not consistent with the rest of the database
  - Changing a node name or number by performing a system generation (SYSGEN)
  - Performing licensed SQLCI2 operations inconsistently throughout the network catalogs
  - Using the CLEANUP utility on portions of a network database, which can leave unresolvable references in other catalogs
  - Loading or copying inconsistent data into a network partition

If any situation arises that you think might affect the network-distributed SQL database, contact your service provider for additional information.

Managing Mixed Versions of SQL/MP

Sometimes a network might be required to run in a SQL/MP mixed-version environment. These activities can produce this situation:

- Running different versions of SQL/MP software simultaneously on separate nodes of an interrelated SQL database application
- Running a newer version of SQL/MP software on a development node that supports a production node or nodes running an older version of SQL/MP software
- Upgrading to a newer version of SQL/MP or downgrading to an older version

Different SQL/MP software versions are not a concern if the nodes involved in a network are not database-interrelated. If your network is operating in one of the preceding scenarios, however, see the *SQL/MP Version Management Guide* for information about software and object version compatibilities in mixed-version networks.

If your network has nodes running C-series and D-series software, processes on D-series nodes must run at a low PIN if they communicate with processes on C-series nodes. For more information, see Mixed-Version Network Considerations on page 2-13.

13 Measuring Performance

During the life of an SQL application, you might need to measure the performance of all, or part, of the application. Several NonStop software products can provide statistical information about performance.

Collecting these statistics requires an in-depth understanding of the system, the layout of the database tables, and the use of the application programs. You usually gather statistics under one of these conditions:

- A benchmark of performance. Typically, you obtain statistics for a benchmark during ideal conditions when all volumes and nodes are available and functioning at peak performance. You obtain these statistics on a finite set of data loaded for the best possible performance.
- A performance problem. Typically, you obtain statistics to determine the cause of a problem. You might have to obtain several samples during different periods of time to compare the results.
- A general sampling. To monitor performance as the SQL database grows and changes, you should periodically obtain a sample of statistics and compare the results against previous samples.
- An equipment change or move requiring database relocation. Whenever the database is moved or changed, you should obtain a performance sampling; the move or change can affect performance.

This section provides an overview of the tools you can use to gather statistics. You can then use these statistics to determine ways of enhancing performance for your application. Enhancing performance is discussed in Section 14, Enhancing Performance.

SQL/MP Tools for Gathering Statistics

Both SQLCI and the SQL programmatic interface have tools for gathering statistics.

The SQLCI commands that display statistics are:

- FILEINFO utility
- SET SESSION STATISTICS ON command
- DISPLAY STATISTICS command

The programmatic data area that receives statistical information is the SQL statistics area (SQLSA).

FILEINFO Utility

The SQLCI FILEINFO utility displays the physical characteristics of SQL tables, indexes, views, collations, and programs. FILEINFO also displays information about Enscribe files. You typically use FILEINFO to display the file label information of files.

For performance statistics, use the FILEINFO utility to determine the index levels and extent and data block use of a table or index. You can improve application performance by effectively using data and index blocks and by effectively using free space in these blocks as follows:

- Index levels

  Index levels are a factor that the SQL compiler analyzes when determining the best access path for a statement, because performance improves as the number of levels decreases. You can obtain the index levels of an index by using the FILEINFO *`index-file-name`*, the DETAIL command, or by querying the INDEXES catalog table containing the description of the index.

- Extent use

  The EXTENTS option of the FILEINFO utility displays information about the number and use of a file's extents. You can use this information to monitor available extents of a file or empty extents.

  For distributed or partitioned tables or indexes, you can determine the extent spread over the partitions.

- Data block use

  The STATISTICS option of the FILEINFO utility displays a map of the data blocks. This information shows used blocks, free blocks, number of records in a file, and slack information. As a file becomes full, the slack and free blocks decrease. With less space, insert and update operations can cause block splits.

For OSS files, these considerations apply:

- You cannot specify an OSS path name as input to the FILEINFO command, but you can specify the Guardian ZYQ name associated with the OSS program.
- The name of an OSS file is displayed in its Guardian file name equivalent and then in its path name format. If there is more than one path name linked to the program, only one path name is displayed (the first path name available to the current user).
- Several informational items are not displayed because they do not apply to OSS files. For example, the EXTENTS option displays a message that EXTENTS information does not apply to an OSS file.
- The owner and security are displayed as OSS.
- The STATISTICS option is equivalent to the DETAIL option.

As an alternative, run FUP or an appropriate OSS utility to obtain information about an OSS file.

SET SESSION STATISTICS and DISPLAY STATISTICS Commands

SQLCI provides the STATISTICS option of the SET SESSION command; this option displays the statistics after each DDL, DML, or DCL statement executed in the session.

You can also use the DISPLAY STATISTICS command to get statistics on a single statement. The DISPLAY STATISTICS command displays statistics for the immediately preceding DDL, DML, or DCL statement.

To obtain statistics, you use either of these commands in your SQLCI session:

```
>> SET SESSION STATISTICS ON;      --Enter before statements
```

```
>> DISPLAY STATISTICS;             --Enter after a statement
```

The statistics displayed after each statement appear in this format, preceded by information about statement execution timing:

```
               Records   Records   Disk  Message Message Lock
Table Name    Accessed      Used  Reads    Count   Bytes   WE
```

Elements of the display follow:

- Table Name is the name of the table for which statistics are being displayed.
- 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 Reads gives a count of the number of disk reads caused by accessing this table.
- 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 gives a count of the message bytes sent to access this table.
- 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.

For example, the DISPLAY STATISTICS command might present this data:

```
Estimated Cost         9

Start Time            89/04/01 13:07:12.822479
End Time              89/04/01 13:07:18.865150
Elapsed Time                   00:00:06.042671
SQL Execution Time             00:00:00.392796

                Records   Records  Disk  Message Message Lock
Table Name     Accessed      Used Reads    Count   Bytes   WE
\a.$b.c.d           123        22     3       10    3245
\w.$x.y.z       9987231         1 99999        1     100    e
\sanfran.$mamoth.longestt.filename
                      1         1     0        1     100    w
```

With these statistics, you can quickly monitor the performance of a specific statement on specific objects. The information provided can help you to:

- Determine the comparative performance of similar objects. For instance, you can determine the effect of a new index on a table compared to the performance without the index, or you can determine the performance after an UPDATE STATISTICS statement.
- Display the statistics of various queries or DML statements.
- Monitor the estimated cost of a compiled statement or an ad hoc query. The larger the estimated cost, the greater the execution time. You can then investigate costly SQL statements for additional indexes, for out-of-date statistics on referenced tables, or for poorly designed queries.

SQL Statistics Area (SQLSA)

The SQL statistics area (SQLSA) is a data area programmers can use to receive statistics after SQL statement execution. To use this area, programmers must include the INCLUDE SQLSA statement in the host language program. When the SQLSA is present, the program passes the data area to the SQL executor; then the executor accumulates and returns statistics.

The DML statements for which statistics are returned are: OPEN CURSOR, FETCH, SELECT, INSERT, UPDATE, and DELETE. Statistics are also returned for prepared DML statements executed with either the EXECUTE or EXECUTE IMMEDIATE statement.

Statistics are kept on a table-by-table basis for a maximum of 16 tables. These statistics include the number of tables accessed, records accessed, records used, number of disk reads, number of wait times for locks, and so forth.

SQLSA statistics also return the total processor time used by all ESPs and sort processes (SORTPROGs). These statistics are useful for queries that use parallel execution plans. They are not kept for each individual table or for each individual ESP or SORTPROG, but rather for all tables and ESP and SORTPROG processes involved in the query.

SQLSA statistics are not cumulative. For example, while a CURSOR is open, the statistics reported apply only to each specific SQL statement issued, such as the OPEN statement and each individual FETCH statement, *not* to the entire set of operations spanning the use of the cursor from open to close. To accumulate statistics for a sequence of operations, you must maintain separate counters and add to them after each SQL statement that affects the SQLSA.

The SQLSA also receives statistics on prepared dynamic SQL statements. These statistics include the number of input and output variables, the length of the buffer required for input and output variables, and the length of a buffer for name maps.

For additional information on using the SQLSA, see the *SQL/MP Programming Manual* for your host language.

Measure Performance Measurement Tool

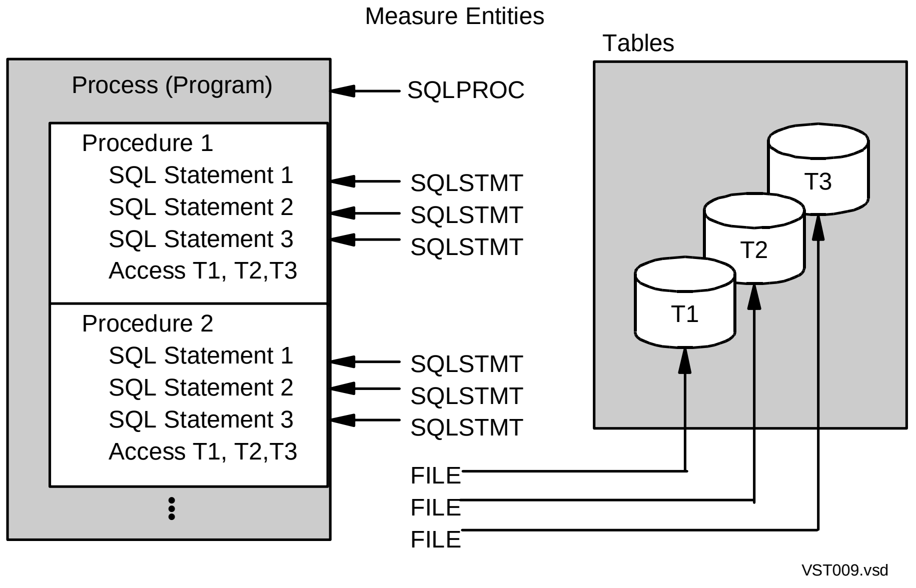
Use the Measure product to collect statistical information on SQL database objects and SQL processes (host language programs with SQL statements) and to generate reports. You select a process for measurement by specifying the process in a Measure ADD command in effect when the process executes.

You can collect performance statistics for SQL/MP objects by using these Measure entities:

- SQLPROC provides information about an SQL process. There is one SQLPROC counter record per SQL process selected.
- SQLSTMT provides information about all SQL statements within an SQL process. There is one SQLSTMT counter record per SQL statement of a selected SQL process.
- FILE allows an SQL database object to be selected for accumulating information about file activities.

Figure 13-1 on page 13-6 illustrates Measure entities and corresponding program structures for SQL processes.

Figure 13-1. Measure Entities and Program Structures



To reduce the cost of overhead for the Measure interface to SQL/MP, the Measure product updates more than one counter per call. The overall cost of using the Measure product depends on the frequency of the intervals for measurements and on the number of active SQL statements. As the number of active SQL statements in a program increases, the cost of performing measurements increases.

The Measure product provides other entities to measure activity on processes, processors, and disk processes. For information on how to set up the Measure product and prepare reports, see the *Measure Reference Manual* and the *Measure User's Guide*.

Statistics and Reports for SQL/MP

You can use the three Measure entities to gather statistics on an SQL database and application programs. After gathering the statistics, you can generate reports about the statistics. The following paragraphs describe the information gathered by the entities.

SQLPROC Statistics

The SQLPROC report provides information on specific statistics concerning recompilations, NEWPROCESS calls, and opens of an SQL process.

You can monitor automatic recompilation time with the SQLPROC report so that you can determine the best compiler option: RECOMPILEALL or RECOMPILEONDEMAND. For a description of these compiler options and their effects on performance, see SQL Compilation and Recompilation on page 10-6.

SQLSTMT Statistics

The SQLSTMT report provides information for specific statements of an SQL process. SQLSTMT entities gather statistics for all statements of a process selected for measurement; there is one SQLSTMT entity for each statement. The SQLSTMT report identifies the SQLSTMT section name for each statement.

In the report, a section name is identified by the procedure *name* and index #*nn*, which relates to the SQL Section Paragraph number generated during the host language compilation. An SQL section is generated for each SQL statement and is listed in the compilation output following the program code. The exception is for the statements on cursors: OPEN, FETCH, and CLOSE cursor statements. The counters of the OPEN, FETCH, and CLOSE cursor statements all contribute to the counter of the DECLARE CURSOR section number.

The SQLSTMT report gathers statistics on busy time, disk reads, sorts, lock waits, timeouts, and message activity. Although the SQLSTMT report can provide you with a large amount of information, you need an in-depth understanding of the program to interpret the statistics correctly.

When enabled, the Measure product allocates the SQLSTMT counter records upon receiving the first call from the SQL executor. This initial call to the Measure product takes slightly longer than subsequent updates. Because the records for each statement are created only when a statement is used, a LIST command on SQL entities returns only allocated records.

You can use the SQLSTMT report for various purposes. For instance, you can relate the SQL statement indexes in the SQLSTMT report to Source Line Table (SLT) indexes that appear on compiler listings, as follows:

- In the case of a COBOL program, compare the SQLSTMT report with the preprocessor listing generated by the COBOL compiler.
    - For each COBOL statement, use the SQL preprocessor to generate a PERFORM SQL DO *n* statement that is actually a call to a subprocedure. This subprocedure, in turn, calls the SQL executor and passes to the executor a parameter named SQLIN*n*.
    - In the preprocessor listing, locate the declaration for SQLIN*n*. Then, in the SQLSTMT report, find the point where the value of the field named SLT-INDEX also equals *n*.

      For example, if you are looking at an SQLSTMT report in which SLT-INDEX has a value of 2, find the place in the preprocessor listing where the SQLIN*n*.SLT-INDEX data structure shows *n* with a value of 2. You can then use this data structure to determine which COBOL statement called PERFORM SQL DO *n*.

  For more information about the SQLIN*n*.SLT-INDEX data structure, see the *SQL/MP Programming Manual for COBOL85*.

- In the case of a C, Pascal, or TAL program, no separate preprocessing occurs, and no preprocessor listing is generated. The source listing produced by the compiler, however, includes comments that show the value of SLT-INDEX. By using this value, you can find the corresponding information in the SQLSTMT report in much the same way as you can with a COBOL program.

FILE Statistics

You can monitor database files with the FILE entity. The FILE report provides information on specific file and record use by a user process. Counters accumulate information for these events: busy time, reads, writes, updates, deletes, records used, message activity, lock waits, timeouts, and escalations of locks. The FILE report can provide you with specific data on SQL tables. You can use the FILE report along with other reports on a specific volume or subvolume.

SQL/MP Measurement Models

When using the Measure product, you must determine whether the overhead for gathering Measure statistics is worth the information provided by the reports. You might find certain statistical information more meaningful with a few samplings. You should, of course, use the Measure product for gathering detailed statistical information for problem analysis.

This subsection describes three types of statistics:

- Startup cost of an application program
- Execution cost of a running process
- Database access costs for SQL tables and indexes

Startup Cost

You can use these counters to analyze the startup cost of an application program. These statistics are gathered by the SQLPROC entity.

- SQL-OBJ-RECOMPILE-TIME contains the elapsed time spent on recompiling an invalid program. The recompile time should be zero when a valid program executes and is not recompiled. (An SQL program is automatically recompiled at run time if the SQL compiler option RECOMPILEALL is specified at explicit compile time and the program has been subsequently invalidated.)
- SQL-NEWPROCESS contains the number of times the SQL compiler or SQL catalog manager was started. The call to NEWPROCESS should be zero when a valid program executes if the program is not automatically recompiled and does not contain any DDL statements.

  For SQLCI or for programs being automatically recompiled, the count should be 2 or fewer. A number higher than 2 indicates that multiple catalog managers might be required on distributed nodes or that compiler or catalog manager timeouts have occurred.

The counter SQL-NEWPROCESS-TIME contains the amount of time spent waiting for the call to NEWPROCESS to complete and is included in the total startup time.

- OPENS contains the number of calls to open tables that were required by this program. The elapsed time spent executing the opens is stored in OPEN-TIME. After an SQL program is started, the files are open and remain open for the duration of the session.

Execution Costs

Use these SQLSTMT counters to analyze the execution costs of a running SQL/MP process. These counters provide information on a statement basis. For counters that have the same names as counters for database access costs (described in the following subsection), you can directly compare the statement values with the table values returned by those counters.

After a program begins running, startup costs have already been incurred. The costs associated with processing the statements are stored in the SQLSTMT entity. The first time a statement in a procedure executes, overhead is added for setting up the counters for the procedure.

You can use these SQLSTMT counters to analyze a running process:

- CALLS stores the number of times the SQL statement was executed.
- ELAPSED-BUSY-TIME stores the wall-clock elapsed time spent on the particular statement. To compute the average elapsed time per call, divide the elapsed busy time by the number of calls.

  Note that the first time a statement in a procedure executes after measurement has been started, a setup time is included for allocating all the SQLSTMT counters for the procedure.
- DISK-READS stores the number of physical disk reads performed for a particular statement.
- RECOMPILES stores the number of times the statement was recompiled. For valid statements, this number should be zero. If the statement has been recompiled, the counter for each session would be 1, because an invalid statement is usually recompiled only once in a session. If this number is 1 or greater, you should consider explicit SQL compiling the program.

  The time spent in recompiling this statement is stored in the ELAPSED-RECOMPILE-TIME value. This value includes the actual compilation time, plus disk read, message, and NEWPROCESS time, involved in initiating the SQL compiler.
- RECORDS-ACCESSED stores the number of records accessed for the statement. If the statement accesses many records but uses only a few, you could create an

index to reduce the number of records searched before returning records that satisfy the query.

- SORTS stores the number of times the external sort process was invoked to return the data in the desired order. A value in this field indicates that the data is not being retrieved in the order supported by a key (primary key or index). The amount of time spent sorting is stored in the ELAPSED-SORT-TIME counter.

  Performance might decrease in proportion to the amount of time spent sorting data. By monitoring the sort time of each statement, you can determine the statements and the associated indexes that might improve performance. An external sort is not invoked if the number of records to sort is fewer than 400.

- TIMEOUTS stores the number of times this statement received a request timeout because of a possible congested disk volume or network. This number should be zero. You should examine the cause for any number greater than zero.
- LOCK-WAITS stores the number of times the statement waited for a lock request. This number should be zero or be quite small. If the number is large for your application, you should examine the cause. Depending on the situation, you might consider a finer locking granularity (for example, row locks instead of generic locks or table locks) or redesigning the database.
- ESCALATIONS stores the number of times a record lock was escalated to a file (table) lock. This number should be zero. If this number is greater than zero, you should consider using a table lock for the program.

Database Access Costs

Use the FILE entity to measure database access costs for SQL tables and indexes. The following counters provide information for analyzing disk processing costs for the database.

These counters provide useful information on SQL database tables. You can use these counters to determine the cost of queries.

- RECORDS-USED stores the number of rows returned to the SQL executor on reads, inserts, writes, updates, or deletes.
- RECORDS-ACCESSED stores the number of rows read by the disk process or file system to return the RECORDS-USED value. RECORDS-ACCESSED should always be the same or greater than RECORDS-USED. The ratio between RECORDS-USED/RECORDS-ACCESSED is the selectivity of the statement.

  A query is most efficient when the number of records used is the same or slightly lower than the number of records accessed. If the number of records accessed is much larger than the number used, the query is accessing many unnecessary rows. You can create an index to improve the selectivity.

- DISK-READS stores the number of physical disk reads performed on the file.

- LOCK-WAITS stores the number of times a call to the disk process waited on locked data.
- TIMEOUTS stores the number of timeouts issued on the file. If the number is greater than zero, the file’s timeout value might be too low, thereby defining an insufficient time.
- ESCALATIONS stores the number of times a record lock was escalated to a file (table) lock.

14 Enhancing Performance

The initial step in achieving maximum performance is providing sufficient hardware to handle the throughput and size of the application database.

In addition to hardware, many factors affect the performance of a database and application. Some factors are system dependent, others are application dependent. The factors discussed in this section are specific performance issues that can arise in an SQL/MP environment after it is in use.

Queries are the basis of a relational database application. You specify queries explicitly by using application-embedded SELECT and CURSOR statements, ad hoc query requests, and report writer selections. You specify queries implicitly by using UPDATE, INSERT/SELECT, and DELETE statements.

The number and type of queries used in an SQL/MP environment influence the performance of the database. For a detailed discussion of how to formulate queries to improve query performance while retrieving the desired output, see the *SQL/MP Query Guide*.

Understanding the Implications of Concurrency

Concurrency is defined as access to the same data by two or more processes at the same time. The degree of concurrency available depends on the purpose of the access, on the access modes in effect, and on whether virtual sequential block buffering (VSBB) is used for the access.

SQL/MP provides concurrent database access for most operations. Control of concurrent access is obtained by using access options, locking options, and (for some DDL operations) the WITH SHARED ACCESS option. These operations can be long-running and are thus subject to contention:

- Creating an index
    - When using the WITH SHARED ACCESS option, CREATE INDEX allows concurrent access by DML statements throughout the entire operation except for the short commit phase of the operation. To maximize concurrent access during index creation, specify the WITH SHARED ACCESS option in your CREATE INDEX statement.
    - Without the WITH SHARED ACCESS option, CREATE INDEX allows concurrent access by DML statements that use SELECT with BROWSE or SHARED access during an initial scan phase, but locks out DML accesses during the remainder of the operation. This is the preferred method if you wish to complete the index creation operation as soon as possible and users do not require concurrent access to the data.

- Moving a partition
    - When using the WITH SHARED ACCESS option, ALTER TABLE PARTONLY MOVE and ALTER INDEX PARTONLY MOVE allow concurrent access by DML statements throughout all the entire operation except for the short commit phase of the operation. To maximize concurrent access while moving a partition, specify the WITH SHARED ACCESS option in your ALTER TABLE statement.

      Note. The WITH SHARED ACCESS option does not support two-way splits.

    - Without the WITH SHARED ACCESS option, ALTER TABLE PARTONLY MOVE and ALTER INDEX PARTONLY MOVE allow concurrent access by DML statements that use SELECT with BROWSE or SHARED access during an initial scan phase, but locks out DML accesses during the remainder of the operation. This is the preferred method if you wish to complete the partition move as soon as possible and users do not require concurrent access to the data.
- Creating a constraint

  CREATE CONSTRAINT allows concurrent access by DML statements that use SELECT with BROWSE or SHARED access during an initial scan phase, but locks out DML accesses during a later update phase.
- Updating statistics

  UPDATE STATISTICS allows concurrent access by DML statements that use SELECT with BROWSE or SHARED access during an initial scan phase, but locks out DML accesses during a later update phase.

For more information about concurrency between DDL and DML operations, see “Concurrency” in the *SQL/MP Reference Manual*. For more information about the WITH SHARED ACCESS option, see the subsection, Minimizing Contention, and the “WITH SHARED ACCESS” entry in the *SQL/MP Reference Manual*.

Minimizing Contention

When creating an index or moving a partition, you can minimize contention by using the WITH SHARED ACCESS option of the CREATE INDEX, ALTER INDEX, or ALTER TABLE statements. For example, this CREATE INDEX statement uses the WITH SHARED ACCESS option:

```
CREATE INDEX EMPL2
  ON EMPL (JOBCODE) CATALOG PERSNL
  WITH SHARED ACCESS
    NAME CR_IND_EMP2
    COMMIT BY REQUEST;
```

The WITH SHARED ACCESS option can also be used in embedded programs.

During a CREATE INDEX...WITH SHARED ACCESS operation, SQL sets the AUDITCOMPRESS option to OFF for the base table. Therefore, during the CREATE INDEX operation the audit trail grows at a faster rate than it does when AUDITCOMPRESS is ON (the default). More audit trail space is needed when the AUDITCOMPRESS option is OFF; the amount depends on the intensity of write activity during the CREATE INDEX operation.

After the CREATE INDEX operation completes, the AUDITCOMPRESS option is set to its original value.

When you specify the WITH SHARED ACCESS option, these steps occur:

1. Initialization and load. SQL reads catalog entries for existing (source) objects involved in the operation and creates the new (target) objects for the operation. SQL then begins copying data from the source objects to the target objects.
2. Audit fix-up. Audit fix-up processes search TMF audit trails for any changes made since the load of the records. If changes are found, the target objects are updated to reflect the changes. For index creation, SQL transforms the data as needed.

   At this point, operation depends on COMMIT options selected with the DDL statement:

   - If a BEFORE or AFTER time was specified, SQL waits for the appropriate time window before starting the next (commit) phase. If the time window has passed, SQL performs as specified by the value of the ONCOMMITERROR clause.
   - If [BY] REQUEST was specified, SQL issues a warning to notify the user that the operation is ready, and waits for the user to respond with a CONTINUE statement. At this point, the user sees a “D>” prompt in the SQLCI session. The user can continue the operation, request a rollback, or enter other SQLCI commands except CATALOG, SYSTEM, VOLUME, EXIT, DEFINE-related commands, or a DDL or utility command against the same object as the ongoing DDL operation.

   While the operation waits, the audit fix-up processes continue reading audit trails and updating target objects.
3. Commit. SQL acquires an exclusive table lock on each source object and searches audit trails for any changes made since the last audit fix-up work. SQL updates the target objects to reflect the changes. Finally, SQL updates file labels and catalog files. At this time, exclusive locks are obtained on the other partitions.

For more detailed information, see the “WITH SHARED ACCESS Option” in the *SQL/MP Reference Manual*.

If you do not specify the WITH SHARED ACCESS option, Step 1 and Step 3 are performed, but the audit trail is not searched in Step 3.

Options Available for WITH SHARED ACCESS

The WITH SHARED ACCESS option supports these options:

- REPORT starts or stops EMS reporting for the operation. Events can be sent to $0 or to an alternate collector.
- NAME specifies an SQL identifier as the name of the operation so that you can identify EMS messages for the operation or identify the operation in a CONTINUE statement.
- COMMIT controls the start time for the final phase of the operation and specifies a timeout period for lock requests and handling of retryable errors during the final phase of the operation. It also specifies whether you want user response before continuing to the commit phase. You can use this clause with the initial DDL request or in a CONTINUE statement.

Considerations

These considerations apply to use of the WITH SHARED ACCESS OPTION:

- To eliminate the interval between the time the DDL operation completes and a new online dump is taken, use the WITH SHARED ACCESS option to take online dumps while the DDL operation proceeds. If an online dump exists for a table (or for the base table of an index) and REPORT is specified, SQL sends an event message to EMS indicating when online dumps can be taken. An operator must use the TMFCOM DUMP FILES command to start these online dumps.
- The audit fix-up process searches audit trails for relevant audit information starting from when the associated DDL operation began. Therefore, audit information must be retained on the system or on backup media until the DDL operation completes. Audit trails should not be automatically deleted before the DDL operation completes. If the audit fix-up process does not find audit files online, the system prompts the operator (on the system console) to restore the audit trails. If there are no backed-up audit trails, the request fails.
- Operator intervention might be necessary in these situations:
    - If the audit fix-up process does not find an audit trail online, operator intervention is needed to restore backed-up audit trails. If the requested audit trail does not exist, the request fails.
    - If an online dump on tape is needed for a newly created partition or index, operator intervention is needed to restore the online dump.
    - If an online dump exists for the table (base table for an index) and REPORT is specified, an event is sent to EMS indicating when online dumps can be taken. Operator intervention is needed to start these online dumps.
- Operations that use WITH SHARED ACCESS typically take considerably longer than equivalent operations without WITH SHARED ACCESS. They do, however, cause less application unavailability, because WITH SHARED ACCESS allows

DELETE, INSERT, and UPDATE access during the operation. The time difference depends largely on the number and length of transactions on the nodes that contain source and target objects for the operation, particularly the number and length of transactions that directly involve source objects for the operation.

- You cannot use the WITH SHARED ACCESS option if one or more of these situations exist:
  - The statement executes within a user-defined TMF transaction.
  - Source objects are not audited.
  - Source or target objects reside on a node running a version older than 315 of SQL/MP software.
  - Source objects reside on a node running TMF software released before the D30 RVU.
  - Source or target objects reside on a node running system software released before the D30.00 (for move partition and serial create index operations) or D30.02 RVU (for split partition, move boundary, merge partition, or parallel create index operations).

Avoiding Contention Between DDL Operations

Only one DDL statement can operate on a given SQL object (or partition of an SQL object) at a time. An error occurs if you attempt to run a DDL statement while another process is executing a DDL statement on the same object.

The specific error depends on the statement involved and the phase of the operation at which the conflict occurs. Common errors for this situation include:

```
12  File in use
40  The operation timed out
73  The table is locked
```

Other Operational Considerations

In general, DDL operations follow a three-part process:

1. Lock the catalog entries exclusively for the objects.
2. Open the objects, read or move data, and close the objects.
3. Lock the file labels exclusively while updating them.

Because the locks used in Step 1 and Step 3 of the operation are exclusive, they have no special priority over other locks that can also be issued on the objects. Therefore, to enable the exclusive locks required by these functions, you might need to manage the application activity as follows:

1. During Step 1, do not compile programs that would require the catalogs involved for update or that refer to the affected objects.
2. During Step 2, you can resume the activity.
3. During Step 3, quiesce the application transaction activity so that locks are not in contention.

These situations can arise during the operation of long-running DDL functions:

- For large tables, audit trail space can be exceeded during the course of the operation, resulting in termination of the operation and backout by the TMF subsystem. This condition is minimized if you allow SQL/MP to manage TMF transactions. HP recommends that you do not initiate a user-defined TMF transaction for long-running DDL and utility operations.
- If the operation cannot acquire the exclusive lock when required, SQL/MP terminates the operation abnormally after a predetermined period of time. Remember that the operation requires the simultaneous availability of all file labels that must be changed in Step 3 of the operation, as described previously. The lock timeout value is currently 60 seconds and cannot be changed.

In a similar way, certain other statements or commands present concurrency issues that can affect the result of the operation. When you are duplicating, backing up, or moving data from one object to another, these functions do not require sustained exclusive access to the source objects; the only exclusive access involved is similar to that required in Step 3 at the end of the function. If you are duplicating a table, however, and you want the target table to contain consistent data, you must consider the implications of concurrency.

You cannot achieve a consistent target table if you refer, in a DUP command, to a source table that is being updated while the duplication is in progress. In such cases, you need to consider stopping transaction activity on the affected tables during the data movement. SQL/MP does not enforce this condition by using locks. So, if you do not procedurally enforce a stable source object, the data in the target object might be inconsistent.

Keeping Statistics Current

SQL/MP provides an UPDATE STATISTICS utility to collect and save statistics on columns and tables. The SQL compiler uses these statistics to help determine the cost of access plans. When you have current statistics for a table, you increase the likelihood that the optimizer chooses an efficient plan.

The UPDATE STATISTICS utility changes the information about a table and its indexes in the catalog so that the information more accurately represents the current content and structure of the table. This information is used by the SQL compiler to determine an access strategy.

The UPDATE STATISTICS statement must be user initiated. SQL/MP does not automatically update statistics during DDL operations or following utility commands such as LOAD.

The statistics that UPDATE STATISTICS collects are:

- Current number of rows in the table
- Byte address of EOF
- Percent of nonempty blocks
- Number of index levels for each index on the table
- Number of unique entries in each column
- Second highest value in each column
- Second lowest value in each column

Knowing When to Update Statistics

You might want to run UPDATE STATISTICS after loading or re-creating a table, after structural changes such as creation of an index, or after significant update activity (growth in database size). Before running UPDATE STATISTICS, however, you should consider these:

- If you experience performance degradation, check for fragmentation of blocks. Use the FILEINFO command with the STATISTICS option set on. If blocks are fragmented, running UPDATE STATISTICS and recompiling the queries will not help; first reload the table online by using the FUP RELOAD command.

  Note that, if an object has the UNRECLAIMED FREESPACE or INCOMPLETE SQLDDL OPERATION attribute set, the FILEINFO STAT results might include extraneous records. For more information, see Altering Partition Attributes on page 7-19.

- Run UPDATE STATISTICS only after a table has been loaded with data. Do not run UPDATE STATISTICS when a table is empty.

- Run UPDATE STATISTICS after creating a new index for a table; otherwise, SQL returns a warning for subsequent operations on the table.
- Do not run UPDATE STATISTICS when the UNRECLAIMED FREESPACE or INCOMPLETE SQLDDL OPERATION attribute is set. The results might be incorrect. For more information, see Altering Partition Attributes on page 7-19.

Other performance issues to consider when you experience reduced response time are:

- The volume containing the table might have heavy disk usage.
- If the table or index is distributed, the network might be rerouted or might have heavy usage.
- The programs might be automatically recompiling, thereby decreasing performance.
- Unusually long ad hoc queries or reports might be reducing response time.
- Programs might be waiting for locked data.

Analyzing the Possible Impact of Running UPDATE STATISTICS

Depending on the size of the table, updating statistics can take longer than you would like; therefore, run UPDATE STATISTICS during off hours when peak performance is not required. You can determine the effect of UPDATE STATISTICS on a production query by bracketing UPDATE STATISTICS and EXPLAIN on the queries in a transaction.

If you want to preserve the existing query execution plan, you must be aware that running UPDATE STATISTICS might cause the optimizer to choose a different plan. UPDATE STATISTICS could, for example, change the access path choices made for queries and programs. Usually, you can improve performance by updating the statistics on a table to reflect the current status. The UPDATE STATISTICS operations, however, might not improve performance, as discussed in these paragraphs:

- UPDATE STATISTICS performs a sampling of rows to determine the statistical information. For very large tables, this procedure can take perhaps 10 to 15 minutes per partition. Because this is a statistical sampling method, the statistics gathered are not exact. Any sampling error, however, should not affect the overall performance of the access method.
- The ALL option of the UPDATE STATISTICS statement specifies that you want statistics updated for all columns. If you do not specify this option, only the columns that make up the primary key and columns that have been specified in any index are updated.
- The UPDATE ALL STATISTICS statement might require additional time to gather information on large tables with many columns. This statement, however, ensures a complete analysis of columns that might be used in queries or indexes. If the

table does not have a very large number of columns, you should probably use the ALL option whenever you update the statistics.

- If you use the NORECOMPILE option of UPDATE STATISTICS, the operation does not invalidate the dependent programs. If you want to take advantage of the new statistics, however, you must explicitly SQL compile the dependent programs.
- If you use the RECOMPILE option (the default), the UPDATE STATISTICS operation invalidates dependent programs so that the programs are automatically recompiled when subsequently used. You should explicitly SQL compile these programs to avoid automatic recompilation.
- When statistics are being updated for a table, T, any DDL or DML operation on T might get a timeout error (SQL error -4066) because T is already opened for exclusive access. During the UPDATE STATISTICS operation, the entry for T in the TABLES catalog table is locked. The catalog is available for other SQL operations; however, if other operations attempt to access the record for T in TABLES, then SQL error -8300 is returned, indicating that the record is locked.

UPDATE STATISTICS writes the new statistical information into the catalog tables. After the statement is performed, you cannot undo it. Subsequent compiles, either explicit or automatic, create a best available query execution plan based on the new statistical information.

Testing UPDATE STATISTICS

Because of the significant effect of running UPDATE STATISTICS, you can try to determine the benefits of the operation before you commit to updating the catalog tables. These steps provide two methods for testing the results of UPDATE STATISTICS: one method is for a test environment, the other for a production environment.

To test UPDATE STATISTICS in a test environment, follow these steps:

1. Test a sample set of queries against the production tables by using DISPLAY STATISTICS to obtain the statistical information.
2. Duplicate the table or tables involved to a test location. For a large database, duplicate a subset of the table or tables involved to a test location.
3. Enter an UPDATE STATISTICS statement.
4. Test the same queries against the tables using DISPLAY STATISTICS to obtain the new statistical information.
5. Determine any improvement in performance.
6. If performance improves, enter the UPDATE STATISTICS on the production database.

To test UPDATE STATISTICS in a production environment, follow these steps:

1. Prepare a sample query from your application. (You should probably use a query used often in your application.)

   This example shows the displayed statistics:

   ```
   >>  SELECT ...;
   >>  DISPLAY STATISTICS;

   Estimated Cost                68

   Start Time                89/03/10 14:26:41.494150
   End Time                  89/03/10 14:27:11.123179
   Elapsed Time                       00:00:29.629029
   SQL Execution Time                 00:00:00.686883

                              Records  Records    Disk
   Table Name                Accessed     Used   Reads   . . .

   \SYS1.$VOL.SALES.ORDERS        49       10       3    . . .
   \SYS1.$VOL.SALES.ODETAIL      246       30     246    . . .
   ```

   (This example shows only the leftmost fields that actually appear in a statistics display. For a complete display, see the *SQL/MP Query Guide*.)

2. Determine the effect of the UPDATE STATISTICS statement by issuing the statement within a user-defined TMF transaction. You can then back out the operation if necessary. In an SQLCI session, do these:

   a. Issue a BEGIN WORK statement; then issue UPDATE STATISTICS with the NO RECOMPILE option.

   b. Use EXPLAIN to see if the new statistics give you the better query execution plan. If the estimated cost is significantly less than the original, the UPDATE STATISTICS statement could improve performance. If the cost is not less, this table probably does not need its statistics updated.

   c. Depending on the EXPLAIN output, you can decide whether to commit the transaction (COMMIT WORK) or back out the transaction (ROLLBACK WORK).

   This example shows the statement and command sequence that rolls back the TMF transaction so that statistics are not updated:

   ```
   >>  BEGIN WORK;          <--Begins the TMF transaction
   >>  UPDATE ALL STATISTICS FOR TABLE $VOL.SALES.ORDERS;
   >>  SELECT ...;
   ```

```
>>  DISPLAY STATISTICS;
>>  ROLLBACK WORK;      <--Rolls back the TMF transaction
```

Note. There is one problem with the preceding scenario: you should not enter the UPDATE STATISTICS statement within a user-defined TMF transaction. If the table is large, the user-defined transaction might cause an error on the TMF audit trails. Normally, UPDATE STATISTICS starts the appropriate number of TMF transactions but does not include the scanning of the table for information within a TMF transaction. If you want to use this sample procedure, make sure the TMF audit trails can handle the workload.

You should only test UPDATE STATISTICS in a lightly loaded system; otherwise, the performance will be influenced by the general system load, which could mask differences in query results.

Running UPDATE STATISTICS

Always specify the NO RECOMPILE option when using UPDATE STATISTICS, for these reasons:

- By default, an UPDATE STATISTICS operation invalidates dependent programs, even if UPDATE STATISTICS is executed within a transaction that is backed out.

  Catalogs are audited; program file labels are not. Because program file labels are not audited, updates to program file labels are not backed out. Consequently, if a transaction is backed out, the program file labels are left in an invalid state while the catalog specifies a valid state.

- To avoid invalidating dependent programs and therefore avoid inconsistencies between the program file label and the catalog. Until you explicitly compile the affected programs, however, they will not use the new statistics.

For a thorough evaluation of access options, include key columns, index columns, and those nonindex columns that participate in predicates. To update statistics for all columns, you must specify UPDATE ALL STATISTICS.

This example updates statistics for primary key columns of the EMPLOYEE table and columns that have been specified in any alternate index on the table:

```
UPDATE STATISTICS FOR TABLE EMPLOYEE NO RECOMPILE;
```

This example requests statistics by reading all rows in the first 50 blocks of each partition of the EMPLOYEE file:

```
UPDATE STATISTICS FOR TABLE EMPLOYEE SAMPLE 50 BLOCKS;
```

You can choose to read the entire table (EXACT option) or a specified number of blocks of each partition (SAMPLE *n* BLOCKS option) for computing statistics. These options help control the amount of time spent calculating statistics. If neither of these options is specified, statistics are collected by reading all rows in partitions smaller than 1,000 blocks and approximately 500 blocks from each partition larger than 1,000 blocks.

If you specify the PROBABILISTIC option, SQL ignores the EXACT and SAMPLE *n* BLOCKS options. The PROBABILISTIC option tells SQL to use an algorithm for computing statistics that gives more accurate results than the algorithm used in earlier product version updates (PVUs) of SQL/MP. Moreover, with the PROBABILISTIC algorithm, SQL computes statistics in parallel on partitioned tables.

Statistics are collected at the table level, except for row count and nonempty block count, which are stored on a partition-by-partition basis. Unique entry count is divided equally among the partitions of a table, with any remainder added to the primary partition.

For more information about the UPDATE STATISTICS statement, see the *SQL/MP Reference Manual*.

Using a Test Database for Emulation

Because the SQL compiler uses statistics stored in the catalog to choose the best access paths, you might want to create test databases that emulate larger or smaller databases.

Normally, you update statistics so that they accurately represent the current content and structure of the database. You can alter the statistics in a test database to emulate a database with different statistics: for example, a production database. In this way, you can test different database structures because the SQL compiler will use the emulated statistics in determining the access path.

Note. Use this technique of altering statistics only in a test environment. Never manually alter the statistics of a production database. Use the UPDATE STATISTICS statement only.

SQL/MP catalogs contain statistical information on the database tables registered in the catalogs. You can create test databases that emulate larger or smaller databases by altering these statistics in the test database.

Obtaining Statistics

To perform valid testing, you must have a database identical to the database you want to test. You should create the database with identical object definitions, but you can use different data. The statistics of the test database should match those of the database you want to test.

To obtain the current statistics of a table, you can query the associated catalog.

These examples show how to obtain statistics on a table:

- The index levels of each index of a table as follows:

```
>>  VOLUME $VOL.SALES;
>>  SELECT TABLENAME, INDEXNAME, INDEXLEVELS
+>      FROM $VOL.SALES.INDEXES
+>      WHERE TABLENAME LIKE  "%$VOL.SALES.ORDERS%";
```

- Statistics on the columns of a table as follows:

```
>>  VOLUME $VOL.SALES;
>>  SELECT TABLENAME, COLNAME, UNIQUEENTRYCOUNT,
+>      SECONDHIGHVALUE, SECONDLOWVALUE
+>      FROM $VOL.SALES.COLUMN
+>      WHERE TABLENAME LIKE "%$VOL.SALES.ORDERS%";
```

- Statistics about file information of the table as follows:

```
>>  VOLUME $VOL.SALES;
>>  SELECT B.TABLENAME, F.EOF, F.NONEMPTYBLOCKCOUNT, B.ROWCOUNT
+>     FROM $VOL.SALES.BASETABS B, $VOL.SALES.FILES F
+>     WHERE B.TABLENAME =  "\SYS.$VOL.SALES.ORDERS" AND
+>            B.FILENAME = F.FILENAME;
```

This example obtains statistics on a table and writes the data on a log file to provide an output listing:

```
>>  LOG STATS CLEAR;
>>  SELECT B.TABLENAME, B.ROWCOUNT, B.STATISTICSTIME,
+>          I.INDEXNAME, I.INDEXLEVELS,
+>          F.EOF, F.NONEMPTYBLOCKCOUNT
+>      FROM $VOL.PERSNL.BASETABS B, $VOL.PERSNL.INDEXES I,
+>           $VOL.PERSNL.FILES F
+>     WHERE B.TABLENAME = "\PHOENIX.$VOL.PERSNL.EMPLOYEE" AND
+>            B.TABLENAME = I.TABLENAME AND
+>            B.TABLENAME = F.FILENAME;
TABLENAME                    ROWCOUNT        STATISTICSTIME
---------------------------- -------------- ---------------
INDEXNAME                    INDEXLEVELS  EOF
---------------------------- -----------  -----------
NONEMPTYBLOCKCOUNT
-------------------

\PHOENIX.$VOL.PERSNL.EMPLOYEE             57
211439149245389562
\PHOENIX.$VOL.PERSNL.EMPLOYEE          2       12288
                 2
\PHOENIX.$VOL.PERSNL.EMPLOYEE             57
211439149245389562
\PHOENIX.$VOL.PERSNL.XEMPDEPT          2       12288
                 2
\PHOENIX.$VOL.PERSNL.EMPLOYEE             57
211439149245389562
\PHOENIX.$VOL.PERSNL.XEMPNAME          2       12288
                 2

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

This example obtains statistics on the columns of the table:

```
>>  SELECT TABLENAME, COLNAME, UNIQUEENTRYCOUNT,
+>      SECONDHIGHVALUE, SECONDLOWVALUE
+>      FROM $VOL.PERSNL.COLUMN
+>      WHERE TABLENAME = "\PHOENIX.$VOL.PERSNL.EMPLOYEE";
TABLENAME                           COLNAME
----------------------------------  -------------------------
UNIQUEENTRYCOUNT     SECONDHIGHVALUE      SECONDLOWVALUE
-------------------  -------------------- -------------------

\PHOENIX.$VOL.PERSNL.EMPLOYEE       EMPNUM
               57  +0000000000000000343   +0000000000000000029
\PHOENIX.$VOL.PERSNL.EMPLOYEE       FIRST_NAME
               50
\PHOENIX.$VOL.PERSNL.EMPLOYEE       LAST_NAME
               53
\PHOENIX.$VOL.PERSNL.EMPLOYEE       DEPTNUM
               11  +0000000000000004000   +0000000000000001500
\PHOENIX.$VOL.PERSNL.EMPLOYEE       JOBCODE
                9  +0000000000000000600   +0000000000000000250
\PHOENIX.$VOL.PERSNL.EMPLOYEE       SALARY
               46  +0000000000138000.40  +0000000000019000.00

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

Altering Statistics

You can alter the statistics in the test database to represent the data and structure of the production database for the queries and programs you want to test.

To update the catalog entries in the test database, you must first set up a licensed SQLCI2 process as described in Appendix A, Licensed SQLCI2 Process. Use the licensed SQLCI2 process to update the associated column in the test catalog with the value obtained in the queries shown previously. These examples show how to update the catalog entries.

This example updates the BASETABS table:

```
>>  UPDATE $VOL.PERSNL.BASETABS
+>      SET ROWCOUNT =  57
+>      WHERE TABLENAME = "\PHOENIX.$VOL.PERSNL.EMPLOYEE";
```

This example updates the INDEXES table. You must update each index entry in the INDEXES table.

```
>>  UPDATE $VOL.PERSNL.INDEXES
+>      SET INDEXLEVELS = 2
+>      WHERE INDEXNAME = "\PHOENIX.$VOL.PERSNL.XEMPDEPT";
```

This example updates the FILES table. You must update the table and each index entry in the FILES table.

```
>>  UPDATE $VOL.PERSNL.FILES
+>      SET EOF = 12288,
+>          NONEMPTYBLOCKCOUNT = 2
+>      WHERE TABLENAME = "\PHOENIX.$VOL.PERSNL.EMPLOYEE";
```

This example updates the DELIV_DATE column. To update the column statistics, you must update each column.

```
>>  UPDATE $VOL.PERSNL.COLUMNS
+>      SET  UNIQUEENTRYCOUNT = 11,
+>           SECONDHIGHVALUE  = 000000000000004000,
+>           SECONDLOWVALUE   = 000000000000001500
+>      WHERE TABLENAME = "\PHOENIX.$VOL.PERSNL.EMPLOYEE" AND
+>            COLNAME = "DEPTNUM";
```

After completing the update operations, do not forget to use the FUP REVOKE or DELETE command on the =_SQL_CI2_*sys* DEFINE to stop using the licensed SQLCI2 process.

After you have altered the statistics, you should be able to test most of the features of queries and programs as if they were running on the real database. With this technique, you cannot test certain locking features that require execution on large tables if your test database does not have large tables. You should explicitly SQL compile the programs to use the new statistics.

Deleting a Test Database

During the development and testing cycle, many test databases can be created on your system. From time to time, you might need to purge obsolete databases or clean up disk volumes. The SQL DROP statement and the SQLCI PURGE and CLEANUP utilities can help you perform such operations.

You can use SQLCI PURGE to purge qualified file-set lists of objects to remove tables, views, indexes, collations, and SQL programs stored in Guardian files. If a test database is described in a single catalog, this combination of commands will remove the objects and the empty catalog:

```
24> SQLCI
>> PURGE *.*.* FROM CATALOG $vol.testcat;
DO YOU WISH TO PURGE THE ENTIRE FILESET
  *.*.* FROM CATALOG $vol.testcat
(Y[ES], N[O], S[ELECT], F[ILES]) ?y
TABLE \sys.$vol.testcat

1 OBJECT(S) PURGED
```

In the PURGE command, $*vol.testcat* is the name of a test catalog.

You can use DROP statements to drop individual objects from a catalog and then drop the catalog. These set of statements demonstrates dropping objects and the catalog:

```
26> SQLCI
>> DROP PROGRAM $vol.objs.prog1;
>> DROP PROGRAM $vol1.objs.prog2;
  .
  .
```

```
>> DROP TABLE $vol.data.table1;
>> DROP TABLE $vol2.data.table2;
 .
 .
>> DROP CATALOG $vol.cat1;
```

Note. To drop an SQL program stored in an OSS file, use the corresponding OSS utility. For more information, see the *Open System Services Shell and Utilities Manual*.

You can use the CLEANUP utility to purge SQL objects (except SQL programs stored in OSS files) and the catalogs in which they are described. Normally, the CLEANUP utility is not recommended for undamaged objects. In cases where a test database is self-contained in a test catalog, however, the CLEANUP utility can be used to purge the objects and the associated catalog.

Be sure not to apply the CLEANUP utility to a catalog in which a production database is described.

This example demonstrates purging objects with the CLEANUP utility:

```
25> SQLCI
>> CLEANUP !
+>     (*.*.* FROM CATALOG $vol.testcat) CATALOGS;
```

In the CLEANUP command, `$`*`vol.testcat`* is the name of a test catalog.

Before deleting any database, consider saving it as a test database for regression tests on modified applications at your site.

Optimizing Index Use

An index on a table provides an alternate access path that differs from the inherent access path (primary key). Indexes improve application performance for data retrieval operations. When compiling a statement, the SQL compiler selects the execution plan for a statement by choosing the best access path to the data. If an index exists, the SQL compiler evaluates using the index.

Indexes can also improve performance by eliminating the need for the disk process to access the underlying table. If the query can be satisfied by the columns contained in the index and the access returns unique rows, the underlying table will not be accessed.

When evaluating whether to use an index or a table scan, SQL compares the number of base table scan I/Os and the I/Os for index access. The use of sequential cache for a scan increases the performance of the scan and increases the likelihood of its use.

Index-only access is faster than a table scan. A sort prevented by index access must be looked at closely, however, because the cost of a scan plus a sort might be less than the cost of index and base table access. For more information about selectivity and cost, see the *SQL/MP Query Guide*.

For more information about indexes, see Determining When to Use Indexes on page 3-16.

Maximizing Parallel Index Maintenance

Indexes are automatically updated whenever a row is inserted into the underlying table and when any key column of the index is changed. Multiple indexes can be updated in parallel. The file system accomplishes parallel index maintenance by issuing asynchronous I/O requests to each disk process serving the indexes. Parallel index maintenance occurs automatically without your having to specify a statement or directive.

To take maximum advantage of parallel index updates, the indexes of a table should reside on separate volumes and should be configured on separate processors to eliminate any contention of parallel operations on indexes serviced by the same disk process. Also, when the indexes cannot use separate volumes and processors, some parallelism is achieved by a single disk process, which can process multiple requests concurrently.

These limitations apply to parallel index maintenance:

- Parallel updating is not performed when a large number of indexes are defined on the same table, although the number of indexes that can be defined on the table and still allow parallel updating is quite large.
- Parallel updating is temporarily suspended when the file system is undoing a transaction that failed.

Managing Cache Memory Size

The disk process uses a buffer in virtual memory to keep copies of the disk blocks that have been accessed most recently. This area of virtual memory is called cache. If the disk process finds a requested block in cache, it can satisfy the request immediately without requesting a physical I/O operation.

Cache size has an important effect on performance. The larger the cache, the more likely it is that a block must be read only once.

To see if cache is operating efficiently, use the STAT option of the PUP LISTCACHE command (D-series only). If CACHE READ HITS are less than 90 percent, consider increasing the cache size. If the ratio of CACHE FAULTS to CACHE CALLS is greater than one percent, consider reducing the cache size, adding more physical memory to the processor, or processing to other processors.

For G-series RVUs, use the SCF INFO DISK, CACHE command to display the disk cache configuration information for the specified disk.

Control cache size by using PUP. For more information on setting the cache size, see the *Peripheral Utility Program (PUP) Reference Manual* (D-series only) and the *SCF Reference Manual for the Storage Subsystem* (G-series only).

Maximizing Disk Process Prefetch Capabilities

SQL can enhance performance by reading blocks of data into cache asynchronously before they are needed. This disk process prefetch operation works best when you request long sequential scans through data or when your access plan has a low selectivity value (as described in the *SQL/MP Query Guide*).

The optimizer requests sequential prefetch for all scan operations expected to read sequentially for more than a few blocks.

When sequential prefetch is used, the disk process attempts to read a group of several consecutive blocks with a single I/O operation. The successive read operations do not have to wait for physical I/O and can be satisfied from cache, in parallel, while the disk process performs other I/O operations. To determine if your query uses sequential prefetch, look for the words `sequential cache` in the EXPLAIN output for the query.

A prefetch operation can be done for all table types, for forward processing, for certain types of operations such as scans, updates and deletes of subsets, and for disk operations using virtual sequential block buffering (described in the *SQL/MP Query Guide*).

To maximize disk process prefetch operations, use:

- Large cache
- Mirrored disks
- Well-organized key-sequenced tables (physical sequence closely maps to logical sequence); the FUP LOAD operation can help reorganize an existing table
- Multiple PINs (for more information, see the NUMDISKPROCESSES sysgen parameter)

To check whether the disk process uses prefetch capabilities for your queries, set statistics on, use the PUP LISTCACHE command (D-series only) and the SCF INFO DISK, CACHE command (G-series only) with the STATISTICS option, and use the Measure DISK and DISKOPEN entities.

Managing File System Double Buffering

SQL can enhance performance by allowing the SQL file system to asynchronously prefetch blocks of data from the disk process. When the file system receives a block of data from the disk process, it asynchronously requests the next block without waiting until it has processed the current block. By the time the file system is ready to process the next block, that block is likely to be in memory already and can be processed immediately. This performance enhancement, called file system double buffering, is especially advantageous when the file system and disk process reside on different processors.

SQL can enable file system double buffering only when the disk process uses virtual sequential block buffering (VSBB) and the SELECT statement specifies browse access. For more information about VSBB, see the *SQL/MP Query Guide*.

SQL automatically uses file system double buffering when this feature will enhance the performance of the query. By default, the optimizer does not use double buffering for scanning the inner table in a nested join or key-sequenced merge join.

This feature requires the file system to use two buffers instead of one. In certain circumstances, this feature could potentially exceed the memory size limit for the process file segment (PFS) assigned to the file system for the process. To avoid memory overflow, SQL automatically disables the use of this feature for any more files if file system memory utilization exceeds 70 percent of the PFS. The disabling of this feature is likely to be temporary; when PFS utilization returns to a level below 70 percent, double buffering is enabled again for newly opened files.

The PFS is an area of real memory used by the file system to store operating system information. The PFS size is dynamic; that is, the file system fills it, as needed, up to the PFS size limit assigned by the operating system.

For example, suppose that a server process opens a large number of SQL tables; the file system uses a portion of the PFS for each open. Suppose further that multiple cursor operations called by the server perform sequential table scans using VSBB. The file system temporarily allocates additional portions of the PFS for each cursor operation; if the file system uses double buffering, it allocates twice the amount of buffer space in PFS memory for each cursor operation. Thus, double buffering can increase the amount of the PFS used by the file system.

If a PFS memory overflow occurs, the system is likely to display error message 31. A file system error 31 occurs when insufficient space is available in the PFS for a file system buffer needed to perform the specified operation. For more information about this error message, see the “File System Errors” section of the *Guardian Procedure Errors and Messages Manual*.

To avoid a PFS memory overflow, take one of these steps:

- Use an SQL DEFINE to lower the PFS utilization threshold at which the SQL file system automatically disables file system double buffering for additional files.
- Increase the PFS size limit.

Using an SQL DEFINE to Manage PFS Utilization

You can change the memory utilization threshold at which SQL disables file system double buffering by setting the DEFINE =_SQL_EXE_DOUBLE_SHUTOFF. For example, by increasing the threshold to 90 percent, you increase the use of double buffering but make it somewhat more likely that the file system will exceed the PFS memory size limit. If you lower the threshold to 50 percent, you decrease the use of double buffering but also reduce the likelihood of a PFS memory overflow.

This example changes the threshold from the default value of 70 percent memory utilization to 50 percent utilization by setting this DEFINE value:

```
>  ADD DEFINE =_SQL_EXE_DOUBLE_SHUTOFF,
                        CLASS MAP, FILE X5
```

This example uses the file name X5. The letter X has no significance; it is used to satisfy the syntax of the FILE parameter. The number 5 specifies a 50 percent PFS utilization threshold. In this DEFINE, the numbers 0 through 10 indicate tenths of the PFS size limit. For example, 2 indicates 20 percent; 9 indicates 90 percent.

Thus, if you set this DEFINE value to 0, the SQL executor directs the file system never to use double buffering. If you set the value to 10, the executor uses double buffering whenever it is specified in a query execution plan.

This DEFINE influences the file system through the SQL executor; the DEFINE affects the use of double buffering at run time. (The optimizer already must have specified double buffering in the query execution plan.)

Changing the PFS Size Limit

You can increase (or decrease) the PFS size limit by using the PFS option in the TACL RUN command or by setting the PFS size with the Binder or nld utility.

The default PFS size for a SQL program is 384 KB. If you SQL compile a program using the NOREGISTER option, the default PFS size is 256 kilobytes (KB). The maximum PFS size allowed by the operating system is one megabyte.

This example uses the Binder CHANGE command to increase the PFS size of a program named MYPROG to one megabyte:

```
1> BIND
@  CHANGE PFS 1048576 BYTES IN MYPROG
@  EXIT
```

This example uses the TACL RUN command to increase the PFS size to one megabyte. The number 512 specifies the number of 2048-byte pages allocated to the PFS:

```
1> RUN MYPROG /PFS 512/
```

For more information about these options, see the *TACL Reference Manual*, *Binder Manual*, or *nld and noft Manual*.

Additional DEFINEs for Managing Double Buffering

Two other DEFINEs allow you to manage other aspects of file system double buffering:

- The DEFINE =_SQL_CMP_DOUBLE_SBB_OFF disables file system double buffering for any query that is SQL compiled while this DEFINE is in effect.
- The DEFINE =_SQL_CMP_DOUBLE_SBB_ON enables (turns on) file system double buffering for scanning the inner table in a nested join or key-sequenced merge join. (The default setting is to disable this feature for an inner table of a join operation.)

The preceding DEFINEs influence SQL compilation; that is, they affect the optimizer's selection of double buffering for a query execution plan.

The EXPLAIN plan shows whether the optimizer has chosen file system double buffering for a query execution plan. The EXPLAIN plan reads as follows:

```
SBB for reads      : Virtual, double buffer
```

Also, when the optimizer requests double buffering for a given plan, the executor does not necessarily use double buffering at run time. Its use depends on memory utilization for the PFS, as described in the preceding paragraphs.

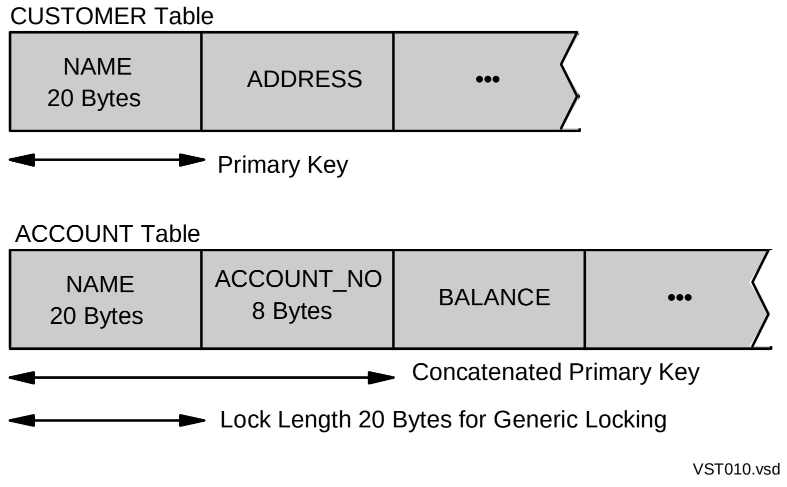
Using Generic Locks

Generic locking is an application-related feature that allows control over the granularity of locking. A generic lock is a lock held by a process on a subset of the rows in a table. Lock granularity is the size of a lockable unit. Generic locking can provide:

- Improved performance, because the application acquires fewer locks while performing operations
- Ability to lock large portions of a table with a single lock without acquiring a table lock
- Reduced risk of a program exceeding the maximum number of locks

Figure 14-1 on page 14-22 illustrates generic locking used in a banking application with the tables CUSTOMER and ACCOUNT. NAME is the primary key for the CUSTOMER table. NAME and ACCOUNT_NO make up the concatenated primary key for the ACCOUNT table.

Figure 14-1. Generic Locking Example



If the lock length of the ACCOUNT table is defined to be the length of the NAME column, SQL/MP acquires locks for an application by using a single lock to lock all rows with the same value for NAME.

If the lock length was not specified when the ACCOUNT table was defined, SQL/MP uses the default lock length, which is the length of the primary key. The default lock length of the ACCOUNT table is 28 bytes, the length of the NAME and ACCOUNT_NO columns. Processing a customer account with the default lock length in effect requires a separate lock for each account belonging to the same customer (a separate lock on each row with the same value for NAME).

To define the lock length for generic locking, you must create a table with key-sequenced organization and a primary key made up of either several columns or an initial column that contains ASCII data. You can specify the LOCKLENGTH attribute either in the CREATE TABLE or CREATE INDEX statement, or in an ALTER TABLE or ALTER INDEX statement for an existing table or index, respectively.

The LOCKLENGTH attribute designates the number of leading bytes of the key that the system should use to identify the rows to lock. All rows with the same value in those leading bytes are locked with a single lock anytime one of those rows is accessed. After you have specified lock length for a table, the length applies to all applications using the table.

The current LOCKLENGTH is stored in the FILES table in the catalog associated with the table of interest.

These are the advantages and disadvantages of generic locks:

- Generic locks improve performance by increasing the granularity of the lock. They decrease concurrency, though, because a lock controls a larger number of rows.

- Generic locks provide a good solution for the application problem in which a table lock is not acceptable, but the application needs so many locks in a transaction that the number might exceed the maximum allowed by the system.

The application problem occurs because the disk process allows a maximum number of locks per process on a partition. An application that examines a large number of rows with the REPEATABLE ACCESS protocol can cause the disk process to escalate row locks to a table lock; however, table locks are not acceptable to many applications.

The application can direct the disk process not to escalate row locks to a table lock by specifying the CONTROL TABLE statement that includes the TABLELOCK OFF option. Using this option, however, the application might generate an error from the disk process if the disk process uses up the control block space for locks. The application can use generic locking to acquire the needed locks with a reduced risk of exceeding the lock limit.

Checking Data Integrity

SQL/MP provides data integrity checking when constraints are defined for a table. When a row is added or altered, the SQL disk process verifies that the new data satisfies any constraints.

Checking data integrity can be performed within program code or with the SQL constraint mechanism. Each method has benefits and performance issues you should consider for your application:

- Data integrity checking in program code
    - Data checking performed in the requester before sending the data to the server is the quickest method of data checking and reduces unnecessary server calls. The requester checks data upon input to ensure the data conforms to certain ranges coded into the requester.
    - Data checking can be performed within an application program or server program before the data is sent to the disk process. This checking reduces unnecessary disk process calls, but still requires the program code to have the data range values.
    - Maintaining programs, requesters, or servers to programmatically check data input can require additional programming time. In addition, your site must have methods or programs to verify that the existing tables conform to the new data checks.
    - Programs with hard-coded validity checking cannot move as easily from one set of users to another as programs without hard-coded values.
- Data integrity checking by constraints
    - Constraints can greatly enhance the flexibility of the programs so that applications move easily from one set of users to another.

- Constraints simplify the change process to a simple, online process. If you add one constraint, the system immediately applies the constraint to all subsequent transactions. The constraint creation process also checks the existing table to ensure that all existing rows conform to the new constraint.
- When SQL verifies the constraints on the input data, the potential message traffic between servers and requesters might be increased when error messages are generated on invalid data.

Evaluate your application to determine the best use of data checking: constraints versus program code. For more information, see Creating Constraints on Data on page 5-51.

Creating Logical Views of Data

Logical views of the database are groupings of data different from the physical database. SQL/MP is efficient for presenting data in logical views; that is, joining tables or other views to create a new window into the data. These logical views can specify only those columns or rows of data that meet the given criteria. SQL/MP returns only the subset of data, if any, that meets the criteria, thereby reducing message data transfer between the disk process and your program.

You can predefine and name logical views with the CREATE VIEW statement, or you can create views logically with a SELECT statement. The performance of these two methods to obtain the same data is equivalent.

For more information, see Creating Views of Base Tables on page 5-38.

Specifying Block Sizes for Files

To achieve maximum performance for sequential or batch operations, use the largest block size for the files underlying your tables. The largest block size, 4096 bytes, is the default size for table and index creation.

There is a locking trade-off, however, when sequential or batch operations run at the same time as online operations. In this case, use a smaller block size to improve OLTP performance.

Specific information on file blocks and computing records contained within those blocks is described under Determining the Number of Records per Block on page 5-15.

Adding and Dropping Partitions

For performance improvement, consider partitioning a table or index to enable them to span multiple volumes or multiple nodes. For more information, see Adding Partitions to Tables and Indexes on page 7-7. As the number of rows in the table or index increases, consider redistributing rows across partitions to balance the distribution of rows. You can use the ALTER TABLE and ALTER INDEX statements to split partitions, move partitions, and move row boundaries.

When all information in a partition becomes obsolete, or when a database design deficiency leaves a partition continually empty, a reference to a table or index defined across this partition results in an unnecessary message to the partition. This message, in turn, results in a correspondingly longer access time to the table or index. In such circumstances, you might want to drop this partition while leaving the others defined for the object intact. For directions on dropping partitions, see Dropping Partitions of Tables and Indexes on page 7-32.

Avoiding Automatic Recompilations

Automatic recompilation can become a significant performance concern. In most cases, you should attempt to be running valid programs at all times to ensure the best possible performance.

Automatic recompilation makes it possible for application programs to continue to perform when invalidating events occur or when access paths are unavailable. The time required to perform the recompilation, however, can noticeably add to the initial response time of the application program that contains the SQL statements.

For more information about automatic recompilation, see SQL Compilation and Recompilation on page 10-6.

Matching Block Split Operation to Table Usage

In a table with key-sequenced organization, when an INSERT operation causes a data block to overflow, the disk process makes room for the new row by splitting the block and transferring some of its contents to a newly allocated block.

The disk process can use one of two methods to split a block:

- Split the block in the middle.
- Split the block at the insertion point when rows are being inserted in sequence and the user has specified the SEQUENTIAL BLOCKSPLIT ON option of the CONTROL TABLE statement.

The SEQUENTIAL BLOCKSPLIT ON option can increase the average number of rows stored per block in certain applications where the disk process cannot detect sequential insertion of rows.

For detailed information about the CONTROL TABLE statement, including the SEQUENTIAL BLOCKSPLIT option, see the *SQL/MP Reference Manual* or SQLCI online help.

Supporting Sort Operations

For certain operations, SQL/MP requires the features of the FastSort sort/merge program. The SQL/MP software requests these services automatically without user interaction or input.

Specifically, SQL/MP calls FastSort during the execution of:

- LOAD commands, when used without the SORTED option to load records into key-sequenced target tables. Enter these commands by using SQLCI.
- CREATE INDEX statements, when used to create indexes on existing nonempty base tables. Enter these statements by using SQLCI or application programs.
- Some queries that require ordering rows in an order different from that of the primary key order or any index. Enter these queries by using SELECT statements in SQLCI or by using cursor operations in application programs.

For an SQL SELECT, DELETE, INSERT, or UPDATE statement, you can determine if a sort occurs by using the EXPLAIN utility, accessed either through SQLCI or the SQL compiler. EXPLAIN reports any sort operations required to run the query.

Although you do not explicitly issue calls to FastSort when using SQL/MP, you can influence the effectiveness of some SQL operations by using =_SORT_DEFAULTS DEFINEs or file-partitioning techniques. In most cases, these techniques are unnecessary; the standard FastSort parameter values are normally sufficient. When working with large tables or indexes, however, you might need these techniques to ensure sufficient disk space or to improve DML statement performance.

You can specify =_SORT_DEFAULTS DEFINEs through your programs at run-time, or you can enter them by using the operating system’s command interpreter or SQLCI. For information about =_SORT_DEFAULTS DEFINE syntax conventions, see the *SQL/MP Reference Manual.* For more information about FastSort, see the *FastSort Manual*.

Specifying Scratch Volumes

When processing input files, FastSort either sorts records in memory or uses one or more scratch files to store intermediate data, as follows:

- For files smaller than 200 KB when the MINTIME option is on, or 100 KB when the AUTOMATIC (default) option is on, the FastSort SORTPROG process performs the entire sort in memory.
- For larger files, SORTPROG uses scratch files to temporarily store intermediate data in groups of records called “runs.” SORTPROG sorts each run, merges the records into an output file, and returns the results to SQL/MP.

You can direct FastSort to use a specific set of volumes for its work. Use the SCRATCH attribute to specify an initial scratch volume. To include or exclude volumes from the pool of volumes FastSort uses once the initial scratch volume is full, use the SCRATCHON and NOSCRATCHON attributes, respectively.

For more information about scratch volumes, see the *FastSort Manual*.

Enhancing Query Performance

You can enhance the performance sorts within SQL queries in several ways. For more information, see the *SQL Query Guide*.

Supporting Parallel Query Execution

When =_SORT_DEFAULTS DEFINEs are used to designate a specific scratch file, the SQL/MP software starts every sort operation with the same SORT DEFINE settings.

If the same scratch file is used during parallel query execution, the first sort request gains exclusive access to the scratch file, and all later sort requests receive an error. To avoid this problem, do not explicitly specify a scratch file by name; instead, specify only the volume name in your =_SORT_DEFAULTS DEFINE and prompt FastSort to create a temporary file. Now, parallel sort operations can take place on the same volume but will access individual scratch files.

The same type of contention problems occur, and also multiply, when you use subsorts to avoid partitioned scratch files. All subsort requests can contend for the same scratch file, and only the first request gets the file. In addition, all sets of subsorts use the same groups of processors. So, if you have eight SQL executor processes, you then have eight sorts, each with subsorts configured in exactly the same way. You must configure parallel subsort operations very carefully.

When using the SQLCI LOAD utility to conduct parallel database loads into a partitioned base table, you can avoid scratch file contention problems if you change the =_SORT_DEFAULTS DEFINE attributes for each load or if you let SQL create scratch files for the operations.

For parallel DDL statement execution (requested by specifying CREATE INDEX...PARALLEL EXECUTION ON), you can identify scratch volumes for the sort processes to use when sorting index records. You identify these volumes in a configuration file whose name you specify in the CONFIG option of the CREATE INDEX statement.

Planning for Temporary File Requirements

When conducting joins and various other operations, SQL/MP creates and uses temporary files. These files exist during the course of an operation and are used for storage during intermediate steps in the operation. When the operation is complete, the temporary files are deleted.

For both serial and parallel operations, the SQL compiler determines the size and location of the necessary temporary files. When it creates a temporary file, the SQL compiler allocates a primary extent of 32 pages and then allocates secondary extents as needed. The secondary extent size is either 512 pages (if SQL estimates the file will be less than 1 GB in size) or 1024 pages (if SQL estimates the file will be larger than 1 GB in size).

The SQL compiler creates each temporary file with a MAXEXTENTS value of 978 extents, permitting a potential maximum length of up to approximately two GBs. Of course, the maximum file length is also constrained by the amount of free space actually available on the disk where the temporary file is stored. This value differs at various run times.

The SQL compiler controls all phases of temporary file allocation automatically. You can, however, use DEFINEs to influence the location and SYNCDEPTH of these files. The =_SQL_TM_*sys_vol* class of system DEFINEs lets you redirect temporary table creation from one volume to another or change the SYNCDEPTH associated with temporary tables. This DEFINE helps avoid file-system error 122, which occurs when a volume becomes full or when DP2 takes over after a processor failure and affects temporary tables created with a SYNCDEPTH of 0 (zero). For more information, see the *SQL/MP Reference Manual*.

To avoid encountering file-system error 43, UNABLE TO OBTAIN DISK FILE SPACE FOR FILE EXTENT, allow enough free space to remain on your disk for ORDER BY, GROUP BY, DISTINCT, and join operations. A good guideline is to keep up to half the total disk space free for these operations.

An additional way to decrease the chance of encountering file-system error 43 is to request parallel execution. When you specify the CONTROL EXECUTOR statement with PARALLEL EXECUTION ON, SQL/MP examines all disks on the system and attempts to spread the temporary files evenly among these disks. This type of balancing promotes the availability of disk space for temporary file extents.

A Licensed SQLCI2 Process

A licensed SQLCI2 process (licensed program) can perform privileged operations, such as deleting or updating rows in catalog tables. Normally only the super ID can perform these operations because of the potential risk to the database. The super ID must explicitly license program files before beginning.

△ **Caution.** These operations can be extremely dangerous to the consistency of the database and the data dictionary. Only the most extreme situations should require the use of a licensed SQLCI2. Only the most knowledgeable SQL/MP manager should attempt to correct problems with a licensed SQLCI2 process.

These operations are restricted to licensed processes:

- Creating or dropping a catalog without reference to the CATALOGS table
- Writing to the SQL catalog tables as if they were user tables (without referring to the SQL file labels)

If the write request is issued from SQLCI, the SQLCI2 process must be licensed. If the statement is issued from a program file, the program file must be licensed.

Licensing SQLCI2

You must license the SQLCI2 process if you want to enable it to perform privileged operations. Running SQLCI2 as the super ID does not pass the SQL license test automatically.

To license a program, the super ID must run the FUP LICENSE command, naming the program. This license persists until the super ID explicitly revokes it by executing the FUP REVOKE command.

To license SQLCI2, do not use the FUP LICENSE command on the $SYSTEM.SYSTEM.SQLCI2 program. Instead, make a copy of SQLCI2 and use that copy for privileged operations.

Running SQLCI2 as SUPER.SUPER

For example, suppose that you want a version of the program that can be used only by the super ID. Suppose that the SQLCI2 program is on $SYSTEM.SYSTEM, and the system catalog is on the default location of $SYSTEM.SQL.

Enter these commands at the command interpreter prompt:

```
33> LOGON SUPER.SUPER, password
34> FUP DUP $SYSTEM.SYSTEM.SQLCI2, $SYSTEM.SYSTEM.SQLCI2L
35> FUP SECURE SQLCI2L, "NN--"
36> SQLCOMP /IN SQLCI2L/ CATALOG $SYSTEM.SQL
37> FUP LICENSE SQLCI2L
```

SQLCI2L is now a licensed version of SQLCI2 and you have secured it for use only by the super ID.

To use the licensed SQLCI2 process, create a DEFINE that enables SQLCI to use the licensed SQLCI2 version rather than the normal SQLCI2 version. This protects you from making unintended changes to your system when you are logged on as super.super.

You must log on as the super ID and create the =_SQL_CI2_*sys* DEFINE, pointing to the licensed version. Use this command:

```
38> ADD DEFINE =_SQL_CI2_sys, CLASS MAP, FILE
$SYSTEM.SYSTEM.SQLCI2L
```

In the command, *sys* is the node (system) name without the backslash.

While this DEFINE is in effect, SQLCI automatically uses the SQLCI2 version in the SQLCI2L file. To stop using the licensed process, you must either end the SQLCI session and delete the DEFINE or log off as the super ID.

This command deletes the DEFINE:

```
48> DELETE DEFINE =_SQL_CI2_sys
```

In the command, *sys* is the node name without the backslash.

Running SQLCI2 as another user

For another example, suppose that you want to set up licensing so that a specific group or specific users can execute it. Select a name for a subvolume on $SYSTEM exclusively for licensed processes, for example $SYSTEM.LICENSED. and copy SQLCI, SQLCI2, and SQLUTIL from $SYSTEM.SYSTEM to this subvolume.

Enter these commands at the command interpreter prompt:

```
33> LOGON SUPER.SUPER, password
34> FUP DUP $SYSTEM.SYSTEM.SQLCI, $SYSTEM.LICENSED.SQLCI
34> FUP DUP $SYSTEM.SYSTEM.SQLCI2, $SYSTEM.LICENSED.SQLCI2
34> FUP DUP $SYSTEM.SYSTEM.SQLUTIL, $SYSTEM.LICENSED.SQLUTIL
36> SQLCOMP /IN SQLCI2L/ CATALOG $SYSTEM.SQL
37> FUP LICENSE $SYSTEM.LICENSED.SQLCI2
37> FUP LICENSE $SYSTEM.LICENSED.SQLUTIL
```

Finally, use FUP or Safeguard to limit who can execute the programs in this subvolume.

Those users can then start SQLCI with this command:

```
RUN $SYSTEM.LICENSED.SQLCI
```

Revoking an SQLCI2 License

To remove the licensed program, you can either revoke the license or purge the program. Either of these commands performs this operation:

```
49> FUP REVOKE $SYSTEM.SYSTEM.SQLCI2L
```

```
49> PURGE $SYSTEM.SYSTEM.SQLCI2L
```

B

Removing SQL/MP From a Node

If you want to install a version of the operating system that does not support SQL/MP, you must remove SQL/MP and all SQL objects from your system (node). A version of the operating system that does not support SQL/MP does not recognize the SQL file structure. You must remove all references to SQL objects before installing that version.

To remove OSS programs, use OSS utilities. For more information, see the *Open System Services Shell and Utilities Reference Manual*.

Using the PUP FORMAT Command to Remove SQL Objects

If you do not need to preserve any data from SQL files, the easiest way to remove SQL/MP is by formatting all the disk volumes:

1. Back up any Enscribe files, EDIT files, and other files to tape by using the DP2FORMAT of BACKUP.
2. Create a new SIT tape.
3. Cold load the new SIT tape by using the $SYSTEM format tape cold-load method of installation. This operation formats $SYSTEM and restores the SIT to $SYSTEM.
4. Complete the INSTALL steps.
5. Use PUP FORMAT and LABEL (D-series only) to format and label all the disk volumes. Use SCF INITIALIZE DISK and ALTER DISK, LABEL (G-series only) to format and label all the disk volumes.
6. Use RESTORE to restore the Enscribe files, EDIT files, and other files that were backed up before the cold load.

Using the CLEANUP Utility to Remove SQL Objects

If you need to retain the data in SQL/MP objects, or if you need to convert SQL tables to Enscribe files, you must carefully follow these steps:

1. Check that the TMF subsystem is operational. Enable all volumes containing SQL objects for TMF auditing.
2. Back up SQL objects that you want to save with the ARCHIVEFORMAT option of BACKUP.

3. For SQL tables containing data you need to preserve for use after removing SQL, create empty Enscribe files. Then, use the SQLCI COPY or LOAD utility to copy the tables into those Enscribe files.

4. Use the CLEANUP utility to purge the SQL objects and the catalogs in which they are described. You must ensure that you do not prematurely apply CLEANUP to the system catalog in which the $SYSTEM.SYSTEM.SQLCI2 program is registered. Therefore, you should either purge each disk volume one at a time or purge all objects except those in the system catalog.

   This CLEANUP command purges all objects except those residing in the system catalog:

   ```
   24> SQLCI
   >> CLEANUP ! (*.*.* EXCLUDE ($vol.syscat.*,
   +>               $SYSTEM.SYSTEM.SQLCI2)), CATALOGS;
   ```

   In this CLEANUP command, $*vol.syscat* is the name of the system catalog. The CATALOGS option purges the catalog tables from the designated disk volume.

5. Purge all the SQL objects from the system catalog except the $SYSTEM.SYSTEM.SQLCI2 program.

   To perform this operation, enter this command:

   ```
   >> CLEANUP $vol.syscat.*  ! EXCLUDE $SYSTEM.SYSTEM.SQLCI2;
   ```

△ **Caution.** In the CLEANUP command, do not use the CATALOGS option; you cannot yet purge the system catalog.

6. Use the DSAP to verify that the only SQL objects existing on your node are the system catalog and $SYSTEM.SYSTEM.SQLCI2. Although you follow the preceding steps precisely, DSAP might identify certain objects of a special type, or shadow labels, that still reside on your disks. These shadow labels are created by the disk process.

   To generate a DSAP report of all SQL objects on each disk, enter command at the command interpreter prompt:

   ```
   25> DSAP volume, SQL
   ```

   In the DSAP command, *volume* is the name of a volume on which SQL objects existed.

   In an OBEY command file, enter the DSAP command once for every volume on which SQL/MP objects exist, and then run the commands from the OBEY command file.

   DSAP lists any SQL catalogs, objects, or programs that you did not remove in the preceding steps. If no SQL objects other than the system catalog and $SYSTEM.SYSTEM.SQLCI2 remain on the node, proceed to Step 8 on page B-3.

7. If any shadow labels remain on the node, however, you must remove them before removing SQL. At the SQLCI prompt, enter:

   ```
   >> CLEANUP  *.*.*, SHADOWSONLY;
   ```

   If any SQL objects other than shadow labels remain on the node, specify the name of each object in a CLEANUP command.

   Repeat Step 6 on page B-2 until DSAP does not list any SQL objects except the system catalog and $SYSTEM.SYSTEM.SQLCI2.

Note. If SQL objects, with the exception of the system catalog and the $SYSTEM.SYSTEM.SQLCI2 program, cannot be removed by Steps 1 to 7, contact your service provider.

---

8. Drop the system catalog. This operation also implicitly drops the $SYSTEM.SYSTEM.SQLCI2 program. This command accomplishes this step:

   ```
   26> SQLCI DROP SYSTEM CATALOG system-catalog;
   ```

   In the DROP SYSTEM CATALOG command, *system-catalog* is the name of the volume and subvolume that contain the system catalog. This command can be entered from SQLCI, provided that SQLCI2 is not running. Alternatively, this command can be entered at any time from TACL, as shown in the preceding example. For more information, see the description of the command in the *SQL/MP Reference Manual*.

9. If you have performed the preceding steps but SQL objects still exist on your node, call your service provider for further help before you load the planned operating system.

   If no SQL objects exist, you have removed SQL/MP from the node successfully. You can now load the operating system that does not support SQL/MP.

C Format 2 Partitions

Format 2 enabling allows SQL/MP to support partitions of a size greater than 2 gigabytes (GB) and up to 1 terabyte (TB). The size of a partition is limited by the size of the single disk upon which it resides.

This appendix discusses planning, migration, fallback, interoperability, and third party considerations for using Format 2 partitions.

Planning for SQL Format 2-Enabled Tables and Format 2 Partitions

Migration of Format 1 enabled tables to Format 2-enabled tables depends on:

- Application requirements
- Logical and physical database design
- Hardware configuration

There are also limits you should consider when planning for SQL Format 2 partitions:

- Partition size continues to be limited by the disk size.
- Maximum extent sizes for Format 2 partitions have been increased.
- The maximum number of extents for Format 2 partitions has been decreased.
- The maximum row size is smaller because of a larger block header and record offset size for Format 2 partitions.
- The maximum number of rows that fit in a block is smaller because of a larger block header and record offset for Format 2 partitions.
- The maximum number of partitions is approximately 10 percent fewer for Format 2-enabled tables when compared to EXTENDED tables.

If your application has large databases and stringent downtime requirements, migration might be more appropriate than conversion. Conversion is a separate offline operation, while migration is a stepped refinement of the database as it moves from being a Format 1 database to one that increasingly uses Format 2 partitions.

Conversion requires these steps:

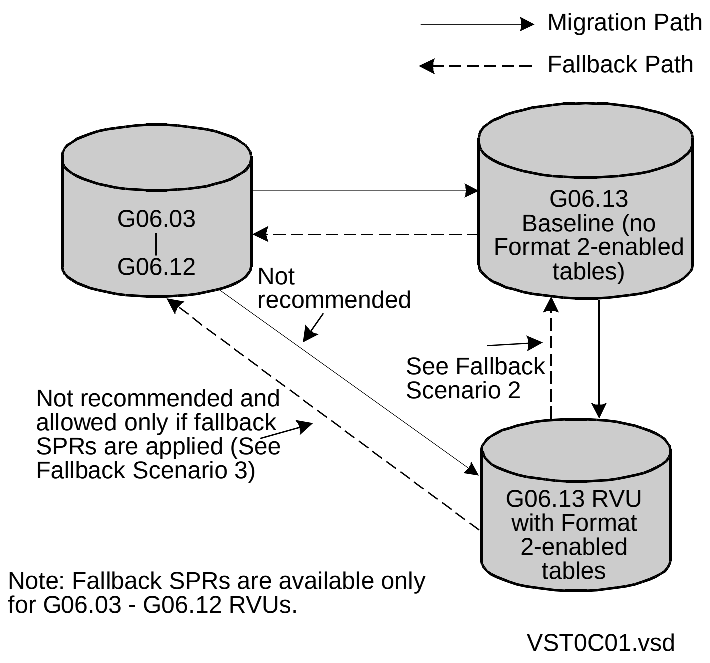
1. Create a copy of the database schema in which requisite tables are Format 2-enabled and partitions of such tables and indexes are Format 2.
2. Stop all your applications and use SQLCI LOAD to copy data from the existing Format 1 enabled tables to the newly created Format 2-enabled tables.
3. Replace the existing tables with each converted table.
4. Restart your applications.

Migration, in contrast, enables you to modify tables a partition at a time to Format 2. Most of the steps involved in this process can be done while your applications continue to access and modify data in the tables.

Whether you attempt conversion or migration of your tables, the steps required have many common elements. Planning is an important part of the process.

Figure C-1 illustrates the possible approaches for migration and fallback planning if your system is currently running a G06.03 through G06.12 RVU.

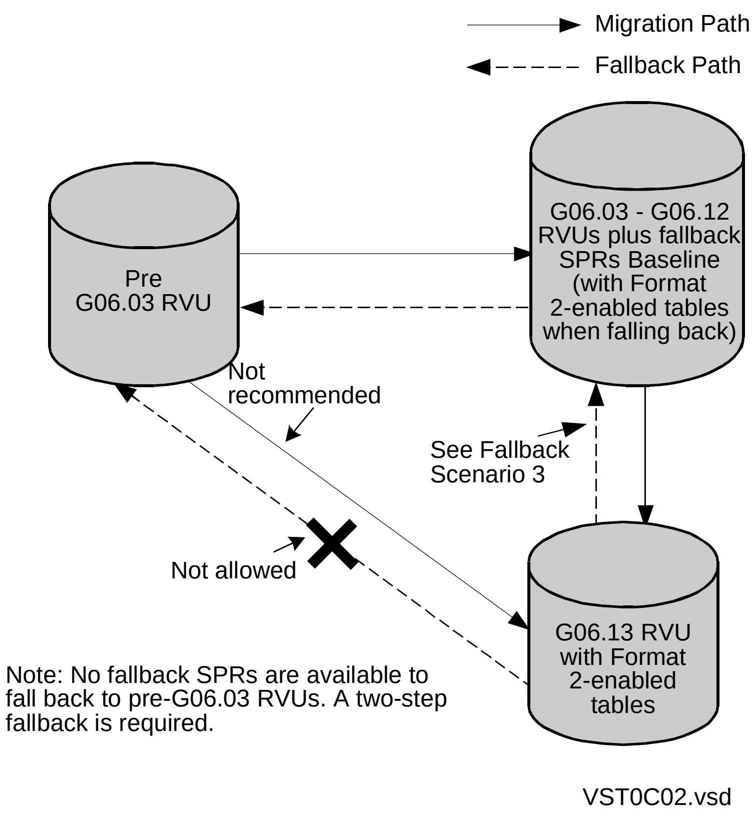
Figure C-1. Migration and Fallback Planning, G06.03 Through G06.12


Migration Path
Fallback Path
G06.03 | G06.12
G06.13 Baseline (no Format 2-enabled tables)
Not recommended
See Fallback Scenario 2
Not recommended and allowed only if fallback SPRs are applied (See Fallback Scenario 3)
G06.13 RVU with Format 2-enabled tables
Note: Fallback SPRs are available only for G06.03 - G06.12 RVUs.
VST0C01.vsd

For more information, see Fallback Scenario 2 on page C-8.

Figure C-2 illustrates the recommended approach for migration and fallback planning if your system is running an RVU prior to G06.03.

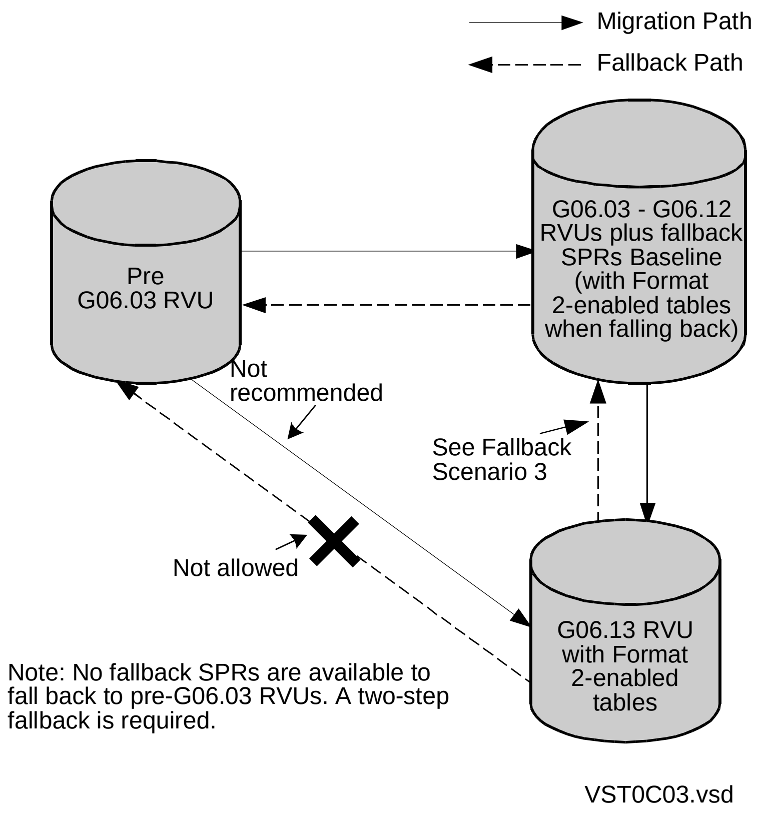
Figure C-2. Migration and Fallback Planning, G06.03 and Earlier RVUs


Migration Path
Fallback Path
Pre G06.03 RVU
G06.03 - G06.12 RVUs plus fallback SPRs Baseline (with Format 2-enabled tables when falling back)
Not recommended
See Fallback Scenario 3
Not allowed
G06.13 RVU with Format 2-enabled tables
Note: No fallback SPRs are available to fall back to pre-G06.03 RVUs. A two-step fallback is required.
VST0C02.vsd

For more information, see Fallback Scenario 3 on page C-9.

Figure C-3 illustrates an alternate approach for migration and fallback planning if your system is running an RVU prior to G06.03. In general, this approach involves slightly more risk than the approach illustrated in Figure C-2 on page C-3, unless all Format 2-enabled tables are cleaned up before falling back. For a description of the required cleanup steps, see Fallback Scenario 2 on page C-8.

Figure C-3. Migration and Fallback Planning, pre G06.03 RVU


Migration Path
Fallback Path
Pre G06.03 RVU
G06.03 - G06.12 RVUs plus fallback SPRs Baseline (with Format 2-enabled tables when falling back)
Not recommended
See Fallback Scenario 3
Not allowed
G06.13 RVU with Format 2-enabled tables
Note: No fallback SPRs are available to fall back to pre-G06.03 RVUs. A two-step fallback is required.
VST0C03.vsd

For more information, see Fallback Scenario 3 on page C-9.

General Planning Considerations

- G06.13 RVU baseline

  Establishing a baseline is an important step prior to migration. HP recommends installing G06.13 RVU and running applications in production prior to using any Format 2-enabled tables. Establishing the baseline is a precaution if issues arise during migration and require temporarily suspending or partially undoing migration activities. Most migrations should be able to proceed as planned; however, not all customer migration scenarios can be tested. As with any software and feature upgrades, you should consider these questions: What can I do if the upgrade runs into problems? How do I fall back to the previous working environment? Establishing a G06.13 RVU baseline separates general G06.13 RVU issues from issues specific to Format 2 partitions or Format 2-enabled tables.

- Fallback baseline

  You might not always be able to establish a G06.13 RVU baseline. For those situations, a set of fallback SPRs are available for G06.03 through G06.12 RVUs. The SPRs allow those earlier RVUs to tolerate the presence of Format 2-enabled tables, as much as possible, and to purge them or any of their associated partitions and views. No operations other than FILEINFO and PURGE are allowed when a Format 2 partition resides on an earlier RVU. Attempts to execute other operations result in appropriate errors being issued. Attempts to perform operations against a Format 2 partition from an earlier RVU where the necessary fallback SPRs have not been installed produce indeterminate results, possibly including data corruption or processor failures. This scenario is also applicable for Format 1 partitions in a Format 2-enabled table.

- Upgrade catalog

  Format 2-enabled tables are version 350 tables and must be registered in version 350 SQL catalogs. Therefore, before you create new Format 2-enabled tables, you must upgrade the catalog where those tables are to be registered to version 350. Before you alter existing tables to be Format 2-enabled, you must upgrade the catalogs where those tables are registered, to version 350. If any Format 2-enabled tables are to be registered in the system catalog, you must upgrade the system catalog to version 350. If you have a separate application catalog, you need not upgrade the system catalog.

- Enable tables

  You can create new Format 2-enabled tables by setting the PARTITION ARRAY value to Format 2-enabled. Similarly, you can alter existing tables by changing their PARTITION ARRAY value to Format 2-enabled. This action implicitly makes the table and all of its indexes Format 2-enabled. All existing partitions are implicitly still Format 1 partitions.

- Partition management

  Use the partition management functions to add new Format 2 partitions and to populate those new partitions with data that currently resides in previously existing Format 1 partitions. You must move partitions serially within a single table and all of its indexes. After locating appropriate disk space for a new Format 2 partition, perform a simple MOVE operation on the Format 1 partition to move it to a new partition on a new disk as a Format 2 partition.

  Using a round-robin approach, you can move the next Format 1 partition to the space vacated by the previously moved partition, and so on until you have moved all partitions that you intend to be Format 2. Alternatively, you can use a double-move space management method. After you move each partition to free disk space, you can move it back to the space vacated by its prior Format 1 location. With this method, the same free disk space is used temporarily as each partition is migrated in turn. This strategy has the disadvantage of doubling the move operations but has the advantage of keeping individual partitions located on their same disk volumes after the moves are completed.

- SQL recompilation

  If you have the Similarity Check feature enabled, no SQL recompilation is required. Parallel Plans might require recompilation.

- Limits evaluation

  You should evaluate the new limits described earlier in this document for the impact on their application. Special consideration should be given to the limits based on the Format 2 block format in addition to the new maximum number of partitions. The Format 2 block format has implications for maximum row size in addition to disk space requirements.

- Partition naming

  The partition management functions used to migrate your data have an impact on partition naming. After moving and merging individual partitions during migration, partitions will have new names, which could have an impact on your application.

- Queries against the SQL catalog

  The FILES catalog table contains new columns, and any queries against it in your applications might require changes.

- Performance

  You should expect similar performance for the same level of parallelism and volume of data when contrasting the identical database with Format 1 and Format 2 partitions. As always, individual results vary, depending upon your application and database. One way to manage performance expectations during migration is to migrate keeping the same level of parallelism and volume of data. Check that all data continues to be accessed from the same processor during all phases of migration. Do not allow growth in the data stored in each partition. This strategy establishes a point in time with which to compare performance prior to migration and enables you to manage the performance impact of future growth separately from migration.

- Network environment

  Users with multiple Expand nodes must upgrade nodes that contain tables and views that span network nodes to the G06.13 RVU and upgrade catalogs to version 350. Version 350 tables and views are inaccessible from systems running software earlier than the G06.13 RVU. The Backup/Restore product is interoperable between the G06.13 (SQL/MP Format 2 partition enabled) and earlier RVUs. The Backup/Restore product reports errors when it attempts to access a Format 2 SQL object from a pre-G06.13 RVU system. If you specify a wild-card file set, Format 2 SQL objects are skipped.

Operational Considerations for SQL Format 2-Enabled Table Use

Before using any Format 2-enabled tables on your system, you should understand the operational, interoperability, and fallback considerations associated with their use. Then, you should formulate a plan that controls their introduction onto your system and minimizes the risk to your data in the various possible fallback scenarios.

The operational considerations for using SQL Format 2-enabled tables are:

- The Format 2-enabled feature can be used only with key-sequenced tables.
- Before you make the changes described in the next two bullets, you might want to take new TMF online dumps or backups of the SQL catalogs and tables involved, to provide additional fallback protection. You should include all the partitions of the involved tables, not just the partitions that will become Format 2, in addition to all of their associated index partitions and views.
- Before you introduce any SQL Format 2-enabled tables to your system, you must upgrade all the SQL catalogs in which their partitions will be registered to version 350.
- After you upgrade the catalogs, you can create new Format 2-enabled tables using the SQLCI CREATE TABLE command, and you can change existing Format 1 enabled tables to Format 2-enabled by using the SQLCI ALTER TABLE command. When you alter a table to Format 2-enabled, its partitions remain Format 1. You can then create new Format 2 partitions or individually data-migrate Format 1 partitions to Format 2, by using the ALTER TABLE command, with or without SHARED ACCESS. The shared access option allows your applications to update the table while this operation takes place.

  When you alter a table to Format 2-enabled, the labels of all its table partitions and of all the associated index partitions and views are changed to version 350. In addition, the partition array structure in the labels of each table partition and index partition is reformatted.

When you alter a partition (table or index) to Format 2, its label is changed to a new format with expanded fields to allow for the larger attribute values that are possible with Format 2 partitions.

- When all changes described in the previous item are made, you should take new TMF online dumps or backups of the tables involved to establish their TMF file recovery or backup protection. Include all the partitions of the involved tables, not just the partitions that were altered to Format 2, in addition to all of their associated index partitions and views and the SQL catalogs that were changed.
- Block headers in Format 2 partitions are larger than those in Format 1 partitions. Because of this, you cannot alter to Format 2 the partitions of some tables whose block size is close to the sum of their row size and the Format 1 block header size. If you attempt to do this, the ALTER TABLE statement fails with SQL error 1221 and file system error 1096.
- If you introduce a Format 2-enabled table that is partitioned across multiple systems, you must first migrate all affected systems to an RVU that supports this feature.
- If you have any programs or third-party products that use TMFARLIB to read the audit trail on your system, it is best to rebind these programs with the version of TMFARLIB provided with RVUs that support SQL Format 2 partitions and to obtain rebound versions of your third-party products.

  You must use rebound versions of these programs or products, however, before you introduce any Format 2-enabled tables that use the AUDITCOMPRESS option on your system. The AUDITCOMPRESS option is the default when a SQL table is created.

Fallback Considerations

This subsection describes three fallback scenarios. If any SQL Format 2-enabled tables have been created on your system, HP recommends that you fall back using Scenario 2, rather than Scenario 3, if at all possible.

Fallback Scenario 1

If you have not created any SQL Format 2-enabled tables on your system and no version 350 SQL catalogs exist, no Format 2 fallback considerations apply, and you are not restricted in your choice of fallback RVU. However, other fallback considerations might still apply because of intervening RVUs that your fallback bypassed or the use of other features introduced in these RVUs.

Fallback Scenario 2

If you have created any SQL Format 2-enabled tables on your system and you are able to perform all these cleanup steps before falling back, no fallback considerations apply, and you are not restricted in your choice of fallback RVU:

1. Convert all Format 2 partitions back to Format 1, with or without SHARED ACCESS, then convert all Format 2-enabled tables back to Format 1 enabled. Alternately, if some Format 2-enabled tables are not important or if their data is first reloaded back into Format 1 enabled tables, you can instead use the SQLCI DROP command to eliminate them.

   Note. Because there might now be additional data in each Format 2 partition, the original Format 1 partition definitions might be too small to accommodate all the data. You might need to convert certain Format 2 partitions back to multiple Format 1 partitions if the original Format 1 partition was nearly full when it was migrated to Format 2.

2. Downgrade all version 350 SQL catalogs on your system to version 345 or lower.
3. Take new TMF online dumps or backups of all the tables and SQL catalogs that were changed in Step 1 and Step 2. Note that you must include all the partitions that were part of Format 2-enabled tables (Format 1 partitions in addition to Format 2) and their associated index partitions and views.
4. Achieve a clean TMF shutdown, with all audited disks up, using the TMFCOM STOP TMF command.
5. If you have any programs or third-party products that use TMFARLIB to read the audit trail on your system, after the fallback, you must ensure that they do not read audit records created before the fallback. Alternately, you must use versions of these programs rebound with a version of TMFARLIB that contains fallback support. (For more information, see Fallback Scenario 3).

However, other fallback considerations might still apply because of intervening RVUs that your fallback bypassed or the use of other features introduced in these RVUs.

Fallback Scenario 3

If you have created any SQL Format 2-enabled tables on your system but you are unwilling or unable to perform all the cleanup steps described in Scenario 2 before falling back, these fallback considerations apply:

- HP strongly recommends that, before you fall back, you find and record the location of all Format 2-enabled tables (including all their Format 1 and Format 2 partitions and all their associated index partitions and views) and all version 350 SQL catalogs on your system. You can find Format 2 partitions of Format 2-enabled tables (but not their Format 1 partitions) by using this command in SQLCI:

  ```
  FILEINFO $*.*.* WHERE SQL AND FORMAT2
  ```

- You can fall back only to a supported fallback RVU (G06.03 through G06.12) to which all the appropriate fallback SPRs have been applied.

  Note. If you are not running a supported fallback RVU and you want to ensure that you can fall back to the RVU from which you migrated (in the scenario where one or more Format 2-enabled tables remain on your system), you must follow a two-step migration plan:

  1. Upgrade or migrate to a supported fallback RVU (with all appropriate fallback SPRs applied) and run that RVU long enough to verify that your important applications still function correctly.
  2. Migrate to the RVU that supports SQL Format 2-enabled tables.

- After falling back, note that:
  - You cannot open, alter, or rename, or issue SQL statements or SQL utility commands against any Format 2-enabled tables (both Format 1 and Format 2 partitions) remaining on the system. The only operations allowed against these tables are GOAWAY and FILEINFO. All other operations are disallowed, and appropriate errors are returned.
  - You can purge partitions of Format 2-enabled tables individually by using the GOAWAY utility, but you cannot downgrade SQL catalogs from version 350.
  - You cannot access any SQL objects registered in any version 350 SQL catalogs remaining on the system. This includes all Format 1 enabled tables registered in them, in addition to Format 2-enabled tables.
  - Because of this restriction, you might want to initially register all new Format 2-enabled tables in a separate version 350 SQL catalog. When a previously existing SQL table is altered to Format 2-enabled, however, you cannot move its registration to the new catalog without re-creating the table.
  - TMF backout and volume recovery will not be able to restore consistency to Format 2-enabled tables on disk (both Format 1 and Format 2 partitions) during TMF startup. Furthermore, TMF is not able to restore their consistency after a subsequent migration back to an RVU that supports Format 2-enabled tables except by using file recovery to a specific position in the audit before the fallback.

    Note. Because of this TMF limitation, HP strongly recommends against falling back if you cannot achieve a clean TMF shutdown (with all disks up), or if you cannot at least close all important Format 2-enabled tables and ensure that all changes pertaining to them have been flushed to those files on disk.

  - The TMFCOM DUMP FILES command will not dump any partitions of Format 2-enabled tables (both Format 1 and Format 2 partitions) or their associated index partitions and views. If you explicitly include any of them by name in the file-set list for this command, TMFDR logs an EMS message, drops them from the set of files being dumped, and continues with the next file. Wild-card file sets in this command also exclude each of the previously mentioned items and continue with the next file, but without indicating that they have been excluded. In both cases, the dump otherwise completes successfully.

- The TMFCOM RECOVER FILES command will not recover any partitions of Format 2-enabled tables (both Format 1 and Format 2 partitions) or their associated index partitions and views from online dumps. If you include any in the file-set list specified for this command (explicitly or by using wild cards), TMFDR logs an EMS message, drops them from the set of files being recovered, and continues with the next file. TMFDR recognizes only that a given file is associated with a Format 2-enabled table (version = 350) when it is about to restore it from the dump.
- BACKUP will not dump any partitions of Format 2-enabled tables (both Format 1 and Format 2 partitions) or their associated index partitions and views. If you explicitly include any of them by name in the qualified file-set list, BACKUP displays an error message and continues with the next file. Wild-card file sets will exclude each of the previously mentioned items from the resulting set of files being dumped, but without indicating that they have been excluded. In both cases, the backup otherwise completes successfully.
- RESTORE will not restore any partitions of Format 2-enabled tables (both Format 1 and Format 2 partitions) or their associated index partitions and views, from backup tapes. If you include any in the qualified file-set list (explicitly or by using wild cards), RESTORE displays an error message and continues with the next file.
- TACL FILEINFO and FUP INFO commands display error 584 for Format 2 partitions (both table and index partitions) of Format 2-enabled tables and continue with the next file in the file-set list. The information for Format 1 partitions (both table and index partitions) of Format 2-enabled tables and any associated views will be displayed normally.
- DSAP will display error 584 if it encounters Format 2 partitions (both table and index partitions) of Format 2-enabled tables and will indicate that they are Lost Free Space. It will then continue reporting on the other files on the disk.
- DCOM will skip Format 2 partitions of Format 2-enabled tables and indexes, and will continue with the other files on the disk. Format 1 partitions (both table and index partitions) will be relocated along with other files to allow the fragments of free space to be collected together.
- If you have any programs or third-party products that read the audit trail on your system, and if they might read any audit created before the fallback, they must be rebound with a version of TMFARLIB that contains fallback support:
    - The version of TMFARLIB supplied with RVUs that support Format 2-enabled tables also contains fallback support. It detects when it is running on an RVU that does not support SQL Format 2 partitions and then functions appropriately. Therefore, if you rebound these programs or third-party products with the new version of TMFARLIB when you migrated, you can continue to use them after falling back.

- Fallback versions of TMFARLIB contain fallback support. If you choose to use this approach, you should rebind before falling back.

Note. TMFARLIB fallback support involves ignoring, and not returning to the caller, audit records generated for DML changes to Format 2 partitions (changes made to data records or blocks). Audit records for DDL changes to Format 2 partitions (changes made to labels) are still returned. Your audit-reading programs might need modifications to tolerate this incomplete representation of the changes made to Format 2 partitions.

Interoperability Considerations

Programs and utilities running on system software product versions earlier than that of G06.13 will not be able to access or manipulate Format 2-enabled tables or their associated indexes or views on G06.13 RVU or later systems. This limitation applies both to SQL commands executed against these version 350 SQL objects from systems that do not support them, in addition to utility commands that attempt to access version 350 objects because they contain wild cards.

Catalog tables themselves are Version 1 SQL objects and are accessible from systems running earlier software product versions. The FILES catalog table has a different schema in a Version 350 catalog than in prior versions. Therefore, queries run against a Version 350 catalog from a system running earlier software product versions will execute, but could encounter unexpected results because of the additional columns and possible values in that table.

If you have any TACL scripts or OBEY command files for these utilities that run on system software of an earlier RVU and access or manipulate files on a system running an RVU that supports SQL Format 2-enabled tables, consider migrating the system running the older RVU to G06.13 (or later.) Alternately, you should ensure that their operation will not be impaired by introducing SQL Format 2-enabled tables on a G06.13 (or later) system on disks viewed by the scripts:

- SQLCI
- TACL
- FUP
- Backup/Restore

In general, the interoperability behavior of these utilities involves:

- Commands that return only the names of files (such as FILES and FILENAMES) return the names of the partitions of Format 2-enabled tables and their associated indexes and views (version 350 objects) without error.
- Other commands return an error (several are possible, depending upon the product and command) for each version 350 object (file) indicating that the software is unable to support access to objects of that version. The name of the file is included in the text describing the error. Processing then continues with the next file in the list or wild-card expansion for that command, or, if that command is finished, with the next command in the OBEY command file or script.

However, there are many different utility commands and many file-list and wild-card possibilities. Some of them could have unanticipated interoperability considerations.

On systems on earlier RVUs, you can restore previously supported types of files from backup tapes that also contain Format 2-enabled tables or their associated indexes or views. Previous RVUs of Restore are able to bypass version 350 files on backup tapes to restore other files on the tape, but they are unable to restore version 350 files.

Third-Party Provider Considerations

If you are a third-party provider of programs that run on HP NonStop S-series systems and any of your programs use TMFARLIB to read the audit trail, these considerations apply:

- When your customers migrate to an RVU that supports SQL Format 2-enabled tables, they will need new versions of your audit-reading programs before they can introduce any Format 2-enabled tables that use the AUDITCOMPRESS option onto their systems. The AUDITCOMPRESS option is the default when a SQL table is created.
- If your audit-reading programs call ARGETFIELDINFO or ARFETCHFIELDVALUE, or if they call ARGETRECADDR for key-sequenced tables, you will need to provide your customers with new versions of these programs that are rebound with the version of TMFARLIB supplied with an RVU that supports Format 2-enabled tables. Programs that are not rebound with the new version of TMFARLIB receive ARE-INTERNAL-ERROR (-1000) when these calls are used against auditcompressed audit records.
- If any of your customers must fall back from an RVU that supports SQL Format 2-enabled tables to one that does not and if they have introduced any Format 2-enabled tables containing any Format 2 partitions onto their system, and if your audit-reading programs might read any audit created before the fallback, they must use versions of your programs bound with a version of TMFARLIB that contains fallback support when they fall back:
    - The version of TMFARLIB supplied with RVUs that support Format 2-enabled tables also contains fallback support. It detects when it is running on an RVU that does not support SQL Format 2 partitions and then functions appropriately. Therefore, your customers can use the same audit-reading programs you provide for their migration when they fall back.

- Fallback versions of TMFARLIB contain fallback support. You could instead provide fallback versions of these programs that are bound with this version of TMFARLIB.

Note. TMFARLIB fallback support involves ignoring, and not returning to the caller, audit records generated for DML changes to Format 2 partitions (changes made to data records or blocks). Audit records for DDL changes to Format 2 partitions (changes made to labels) are still returned. Your audit-reading programs might need modifications to tolerate this incomplete representation of the changes made to Format 2 partitions.

Index