

HP NonStop SQL/MP Reference Manual

Abstract

This manual describes HP NonStop™ SQL/MP, the HP relational database management system that uses SQL to describe and manipulate data in a NonStop SQL/MP database. The manual includes information about SQLCI, the conversational interface to NonStop SQL/MP.

Product Version

NonStop SQL/MP G07 and H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, G06.27 and all subsequent G-series RVUs, and D46.00 and all subsequent D-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
523352-013	April 2013

Document History

Part Number	Product Version	Published
523352-009	NonStop SQL/MP G07 and H01	July 2008
523352-010	NonStop SQL/MP G07 and H01	August 2008
523352-011	NonStop SQL/MP G07 and H01	November 2008
523352-012	NonStop SQL/MP G07 and H01	August 2010
523352-013	NonStop SQL/MP G07 and H01	April 2013

Legal Notices

© Copyright 2013 Hewlett-Packard Development Company L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Export of the information contained in this publication may require authorization from the U.S. Department of Commerce.

Microsoft, Windows, and Windows NT are U.S. registered trademarks of Microsoft Corporation.

Intel, Itanium, Pentium, and Celeron are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java is a U.S. trademark of Oracle and/or its affiliates.

Motif, OSF/1, UNIX, X/Open, and the "X" device are registered trademarks and IT DialTone and The Open Group are trademarks of The Open Group in the U.S. and other countries.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

© 1990, 1991, 1992, 1993 Open Software Foundation, Inc. This documentation and the software to which it relates are derived in part from materials supplied by the following:

© 1987, 1988, 1989 Carnegie-Mellon University. © 1989, 1990, 1991 Digital Equipment Corporation. © 1985, 1988, 1989, 1990 Encore Computer Corporation. © 1988 Free Software Foundation, Inc. © 1987, 1988, 1989, 1990, 1991 Hewlett-Packard Company. © 1985, 1987, 1988, 1989, 1990, 1991, 1992 International Business Machines Corporation. © 1988, 1989 Massachusetts Institute of Technology. © 1988, 1989, 1990 Mentat Inc. © 1988 Microsoft Corporation. © 1987, 1988, 1989, 1990, 1991, 1992 SecureWare, Inc. © 1990, 1991 Siemens Nixdorf Informationssysteme AG. © 1986, 1989, 1996, 1997 Sun Microsystems, Inc. © 1989, 1990, 1991 Transarc Corporation.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California.

Printed in the US

HP NonStop SQL/MP Reference Manual

[Index](#)[Examples](#)[Tables](#)

[Legal Notices](#)

[What's New in This Manual](#) xxix[Manual Information](#) xxix[New and Changed Information](#) xxx[About This Manual](#) xxxiii[Related Manuals](#) xxxiii[Notation Conventions](#) xxxv[HP Encourages Your Comments](#) xxxvii

[A.](#)

[Access Options](#) A-1[Access Options on DML Statements](#) A-1[Access Options on DDL Statements](#) A-2[Summary: Effect of Access Options on Concurrency](#) A-3[ADD DEFINE Command](#) A-4[Considerations—ADD DEFINE](#) A-5[Examples—ADD DEFINE](#) A-5[AGGREGATE Functions](#) A-6[Alias](#) A-6[ALLOCATE File Attribute](#) A-7[Considerations—ALLOCATE](#) A-7[ALTER CATALOG Statement](#) A-8[Considerations—ALTER CATALOG](#) A-8[Example—ALTER CATALOG](#) A-9[ALTER COLLATION Statement](#) A-10[Considerations—ALTER COLLATION](#) A-10[Examples—ALTER COLLATION](#) A-11[ALTER DEFINE Command](#) A-11[Considerations—ALTER DEFINE](#) A-12[Example—ALTER DEFINE](#) A-12[ALTER INDEX Statement](#) A-13

A. (continued)

Considerations—ALTER INDEX	A-19
Examples—ALTER INDEX	A-26
ALTER PROGRAM Statement	A-27
Considerations—ALTER PROGRAM	A-28
Example—ALTER PROGRAM	A-29
ALTER TABLE Statement	A-29
Considerations—ALTER TABLE	A-37
Examples—ALTER TABLE	A-48
ALTER VIEW Statement	A-50
Considerations—ALTER VIEW	A-51
Example—ALTER VIEW	A-52
APPEND Command	A-52
Considerations—APPEND	A-53
APPENDCANCEL Command	A-56
Considerations—APPENDCANCEL	A-57
APPENDRESTART Command	A-58
Considerations—APPENDRESTART	A-59
AS Clause	A-60
Decorations	A-64
Examples—AS	A-65
AS DATE/TIME Clause	A-67
Examples—AS DATE/TIME	A-69
ASCII Character Set	A-70
AUDIT File Attribute	A-74
Considerations—AUDIT	A-74
AUDITCOMPRESS File Attribute	A-75
Considerations—AUDITCOMPRESS	A-75
Audited Tables	A-76
AVG Function	A-76
Considerations—AVG	A-77
Examples—AVG	A-77

B.

BACKUP Utility	B-1
BASSETABS Table	B-1
BEGIN DECLARE SECTION Directive	B-2
Examples—BEGIN DECLARE SECTION	B-2
BEGIN WORK Statement	B-3

B. (continued)

Example—BEGIN WORK	B-3
BETWEEN Predicate	B-4
Considerations—BETWEEN	B-4
Examples—BETWEEN	B-4
BLOCKSIZE File Attribute	B-5
Considerations—BLOCKSIZE	B-5
BREAK FOOTING Command	B-6
Considerations—BREAK FOOTING	B-6
Example—BREAK FOOTING	B-7
BREAK ON Command	B-8
Considerations—BREAK ON	B-8
Examples—BREAK ON	B-9
BREAK TITLE Command	B-10
Considerations—BREAK TITLE	B-10
Example—BREAK TITLE	B-11
BUFFERED File Attribute	B-11
Considerations—BUFFERED	B-11

C.

CANCEL Command	C-1
Consideration—CANCEL	C-1
Example—CANCEL	C-1
CASE Expression	C-2
Considerations—CASE Expression	C-3
Examples—CASE Expression	C-4
CAST Function	C-4
Valid Considerations—CAST	C-5
Examples—CAST	C-6
CATALOG Command	C-7
Considerations—CATALOG	C-7
Example—CATALOG	C-7
Catalogs	C-8
Operations on Catalog Tables	C-9
CATALOGS Table	C-11
CENTER_REPORT Option	C-12
Consideration—CENTER_REPORT	C-12
Example—CENTER REPORT	C-12
Character Data Types	C-13

C. (continued)

Character Expressions	C-14
Considerations—Character Expressions	C-15
Examples—Character Expressions	C-17
Character Sets	C-17
ISO 8859 Character Sets	C-18
Kanji Character Set	C-18
KSC5601 Character Set	C-19
CHAR_LENGTH Function	C-20
Considerations—CHAR_LENGTH Function	C-20
Examples—CHAR_LENGTH Function	C-20
CLEANUP Command	C-21
Considerations—CLEANUP	C-22
CLEANUP Exception Cases	C-24
Example—CLEANUP	C-25
CLEARONPURGE File Attribute	C-26
Considerations—CLEARONPURGE	C-26
CLOSE Statement	C-26
Considerations—CLOSE	C-27
Example—CLOSE	C-27
Clustering Keys	C-28
Example—CLUSTERING KEYS	C-28
COLLATE Clause	C-29
Collation Definitions	C-30
Comment and Escape Characters in Collation Definitions	C-30
The LC_COLLATE Section of a Collation Definition	C-32
The LC_CTYPE Section of a Collation Definition	C-35
The LC_TDMCODESET Section of a Collation Definition	C-37
Considerations—Collation Definitions	C-38
Examples—Collation Definitions	C-38
Collations	C-43
Column Identifier	C-44
Example—Column Identifiers	C-44
Columns	C-45
COLUMNS Table	C-46
COMMENT Statement	C-48
Considerations—COMMENT	C-48
Examples—COMMENT	C-49
Comments	C-50

C. (continued)

Example—Comments	C-50
COMMENTS Table	C-51
COMMIT Option	C-52
Considerations—COMMIT Option	C-55
Example—COMMIT Option	C-55
COMMIT WORK Statement	C-56
Considerations—COMMIT WORK	C-57
Example—COMMIT WORK	C-57
Comparison Predicate	C-58
Considerations—Comparison Predicate	C-58
Examples—Comparison Predicate	C-60
COMPUTE_TIMESTAMP Function	C-62
Considerations—COMPUTE_TIMESTAMP	C-62
Example—COMPUTE_TIMESTAMP	C-63
CONCAT Clause	C-63
Considerations—CONCAT	C-64
Example—CONCAT	C-64
Concurrency	C-65
Effect of VSBB on Concurrency	C-68
Constraints	C-69
CONSTRNT Table	C-70
CONTINUE Statement	C-70
Considerations—CONTINUE	C-71
Examples—CONTINUE	C-72
CONTROL EXECUTOR Directive	C-73
Considerations—CONTROL EXECUTOR	C-73
Example—CONTROL EXECUTOR	C-73
CONTROL QUERY Directive	C-74
Considerations—CONTROL QUERY	C-76
Examples—CONTROL QUERY	C-76
CONTROL TABLE Directive	C-77
Considerations—CONTROL TABLE	C-85
Examples—CONTROL TABLE	C-92
CONVERT Command	C-94
CONVERT Behavior	C-99
Enscribe Files and DDL Record Definitions	C-100
DDL Primary Keys and Alternate Keys	C-100
DDL Clause Mapping	C-101

C. (continued)

Conversion of DDL Elementary Items	C-103
DDL Groups	C-107
Physical File Attributes of Tables and Indexes	C-107
Partition Attributes of Tables and Indexes	C-108
Examples—CONVERT	C-108
CONVERTTIMESTAMP Function	C-113
Example—CONVERTTIMESTAMP	C-113
COPY Command	C-113
Considerations—COPY	C-122
Enscribe Field Formats	C-126
Field Conversions	C-126
Examples—COPY	C-127
Correlation Names	C-128
COUNT Function	C-130
Considerations—COUNT	C-130
Example—COUNT	C-130
CPRLSRCE Table	C-131
CPRULES Table	C-131
CREATE CATALOG Statement	C-132
Considerations—CREATE CATALOG	C-134
Examples—CREATE CATALOG	C-135
CREATE COLLATION Statement	C-137
Consideration—CREATE COLLATION	C-138
Example—CREATE COLLATION	C-138
CREATE CONSTRAINT Statement	C-139
Considerations—CREATE CONSTRAINT	C-140
Examples—CREATE CONSTRAINT	C-141
CREATE INDEX Statement	C-142
Considerations—CREATE INDEX	C-147
Examples—CREATE INDEX	C-150
CREATE SYSTEM CATALOG Command	C-152
Considerations—CREATE SYSTEM CATALOG	C-152
Examples—CREATE SYSTEM CATALOG	C-153
CREATE TABLE Statement	C-154
Considerations—CREATE TABLE	C-161
Examples—CREATE TABLE	C-162
CREATE VIEW Statement	C-166
Considerations—CREATE VIEW	C-169

C. (continued)

- [Examples—CREATE VIEW](#) C-171
- [CURRENT Function](#) C-173
 - [Example—CURRENT](#) C-173
- [CURRENT_TIMESTAMP Function](#) C-174
 - [Considerations—CURRENT_TIMESTAMP](#) C-174
 - [Example—CURRENT_TIMESTAMP](#) C-174
- [Cursors](#) C-175
 - [Cursor Position](#) C-176
 - [Cursor Stability](#) C-176
- [C89](#) C-176

D.

- [Data Dictionary](#) D-1
- [Data Types](#) D-1
- [DATE Data Type](#) D-8
 - [Example—DATE Data Type](#) D-8
- [DATE_FORMAT Option](#) D-8
 - [Example—DATE_FORMAT](#) D-8
- [Date-Time Data Types](#) D-9
- [Date-Time Functions](#) D-10
- [Date-Time Literals](#) D-10
 - [Examples—Date-Time Literals](#) D-13
- [DATEFORMAT Function](#) D-14
 - [Example—DATEFORMAT](#) D-14
- [DATETIME Data Type](#) D-15
 - [Considerations—DATETIME DATA TYPE](#) D-15
 - [Example—DATETIME](#) D-16
- [DAYOFWEEK Function](#) D-17
 - [Example—DAYOFWEEK](#) D-17
- [DCL \(Data Control Language\) Statements](#) D-18
- [DCOMPRESS File Attribute](#) D-18
 - [Considerations—DCOMPRESS](#) D-18
- [DDL \(Data Definition Language\) Statements](#) D-20
 - [DDL Statements](#) D-20
- [Deadlocks](#) D-22
- [DECIMAL_POINT Option](#) D-22
 - [Considerations—DECIMAL_POINT](#) D-22
 - [Examples—DECIMAL_POINT](#) D-23

D. (continued)

<u>DECLARE CURSOR Statement</u>	D-23
<u>Considerations—DECLARE CURSOR</u>	D-24
<u>Examples—DECLARE CURSOR</u>	D-25
<u>DEFAULT Clause</u>	D-26
<u>Example—DEFAULT</u>	D-27
<u>DEFINES</u>	D-27
<u>Using DEFINES</u>	D-28
<u>Using DEFINES From SQLCI</u>	D-30
<u>Using DEFINES With SQL Programs</u>	D-31
<u>DEFINE Attributes</u>	D-32
<u>DEFINES of Class CATALOG</u>	D-33
<u>DEFINES of Class DEFAULT</u>	D-33
<u>DEFINES of Class MAP</u>	D-33
<u>Summary of DEFINE Attributes</u>	D-34
<u>Examples—DEFINES Used With SQL Programs</u>	D-37
<u>DELETE DEFINE Command</u>	D-38
<u>Examples—DELETE DEFINE</u>	D-38
<u>DELETE Statement</u>	D-39
<u>Considerations—DELETE</u>	D-40
<u>Examples—DELETE</u>	D-40
<u>DESCRIBE INPUT Statement</u>	D-41
<u>Example—DESCRIBE INPUT</u>	D-43
<u>DESCRIBE Statement</u>	D-43
<u>Example—DESCRIBE</u>	D-45
<u>Detail Alias</u>	D-46
<u>DETAIL Command</u>	D-47
<u>Considerations—DETAIL</u>	D-50
<u>Examples—DETAIL</u>	D-51
<u>DISPLAY STATISTICS Command</u>	D-52
<u>Considerations—DISPLAY STATISTICS</u>	D-52
<u>Example—DISPLAY STATISTICS</u>	D-53
<u>DISPLAY USE OF Command</u>	D-54
<u>Considerations—DISPLAY USE OF</u>	D-54
<u>Examples—DISPLAY USE OF</u>	D-56
<u>DML Statements</u>	D-58
<u>DOWNGRADE CATALOG Command</u>	D-58
<u>Considerations—DOWNGRADE CATALOG</u>	D-59
<u>Examples—DOWNGRADE CATALOG</u>	D-60

D.

(continued)DOWNGRADE SYSTEM CATALOG Command	D-61
Considerations—DOWNGRADE SYSTEM CATALOG	D-61
Example—DOWNGRADE SYSTEM CATALOG	D-62
DROP Statement	D-63
Considerations—DROP	D-63
Example—DROP	D-66
DROP SYSTEM CATALOG Command	D-66
Considerations—DROP SYSTEM CATALOG	D-66
Examples—DROP SYSTEM CATALOG	D-67
DSL Statements	D-67
DSLACK File Attribute	D-68
Considerations—DSLACK	D-68
DUP Command	D-69
Considerations—DUP	D-74
Examples—DUP	D-78
Dynamic SQL	D-80
Summary of Dynamic SQL Statements	D-80
Determining When to Use Dynamic SQL	D-80
Features of Dynamic SQL	D-81

E.

EDIT Command	E-1
Examples—EDIT	E-1
Embedded SQL	E-2
END DECLARE SECTION Directive	E-3
Example—END DECLARE SECTION	E-3
ENV Command	E-3
Considerations—ENV	E-3
Example—ENV	E-4
ERROR Command	E-4
Examples—ERROR	E-5
Error Messages	E-6
EXECUTE Statement	E-7
Considerations—EXECUTE	E-9
Examples—EXECUTE	E-10
EXECUTE IMMEDIATE Statement	E-11
Considerations—EXECUTE IMMEDIATE	E-11
Example—EXECUTE IMMEDIATE	E-11

E. (continued)

EXISTS Predicate	E-12
Examples—EXISTS	E-12
EXIT Command	E-13
Example—EXIT	E-13
EXPLAIN Directive	E-13
Considerations—EXPLAIN	E-13
Examples—EXPLAIN	E-20
Expressions	E-21
Numeric, Date-Time, and Interval Expressions	E-22
Considerations—Expressions	E-23
Examples—Expressions	E-26
EXTEND Function	E-28
Considerations—EXTEND	E-29
Examples—EXTEND	E-29
EXTENT File Attribute	E-30
Considerations—EXTENT	E-31

F.

FC Command	F-1
Considerations—FC	F-1
Examples—FC	F-2
FETCH Statement	F-3
Considerations—FETCH	F-4
Examples—FETCH	F-4
File Attributes	F-6
File Attributes of SQL Objects	F-7
File Organizations	F-8
FILEINFO Command	F-9
Considerations—FILEINFO	F-11
BRIEF Display for SQL Objects and Guardian Files	F-11
DETAIL Display for Objects (Except Views) and Guardian Files	F-14
DETAIL Display for Views	F-20
BRIEF and DETAIL Display for OSS Files	F-21
EXTENTS Display	F-21
STATISTICS Display	F-22
Examples—FILEINFO	F-22
FILENAMES Command	F-25
Example—FILENAMES	F-25

F. (continued)

- [FILES Command](#) F-26
 - [Example—FILES](#) F-26
- [FILES Table](#) F-27
- [Filesets](#) F-29
 - [Examples—Filesets](#) F-29
- [FORMAT File Attribute](#) F-30
 - [Considerations—FORMAT](#) F-30
- [FREE RESOURCES Statement](#) F-30
 - [Considerations—FREE RESOURCES](#) F-31
 - [Examples—FREE RESOURCES](#) F-31
- [Functions](#) F-32
- [FUP Command](#) F-33
 - [FUP Commands and SQL Objects](#) F-33
 - [Considerations—FUP](#) F-35
 - [Examples—FUP](#) F-35

G.

- [Generalized Owner](#) G-1
- [GET CATALOG OF SYSTEM Statement](#) G-1
 - [Considerations—GET CATALOG OF SYSTEM](#) G-1
 - [Examples—GET CATALOG OF SYSTEM](#) G-2
- [GET VERSION Statement](#) G-2
 - [Consideration—GET VERSION](#) G-3
 - [Examples—GET VERSION](#) G-3
- [GET VERSION OF PROGRAM Statement](#) G-4
 - [Consideration—GET VERSION OF PROGRAM](#) G-4
 - [Examples—GET VERSION OF PROGRAM](#) G-5
- [GOAWAY Command](#) G-6
 - [Considerations—GOAWAY](#) G-6
 - [Examples—GOAWAY](#) G-6
- [Group Manager](#) G-7
- [Guardian Names](#) G-7
 - [Considerations—Guardian Names](#) G-8
 - [Example—Guardian Names](#) G-8

H.

<u>HEADING Clause</u>	H-1
<u>Consideration—HEADING</u>	H-1
<u>Example—HEADING</u>	H-1
<u>HEADINGS Option</u>	H-2
<u>Example—HEADINGS</u>	H-2
<u>HELP Command</u>	H-2
<u>Considerations—HELP</u>	H-3
<u>Examples—HELP</u>	H-3
<u>HELP TEXT Statement</u>	H-4
<u>Consideration—HELP TEXT</u>	H-4
<u>Example—HELP TEXT</u>	H-4
<u>HISTORY Command</u>	H-5
<u>Example—HISTORY</u>	H-5
<u>Host Identifiers</u>	H-5
<u>Host Programs</u>	H-6
<u>Host Variables</u>	H-6

I.

<u>ICOMPRESS File Attribute</u>	I-1
<u>Considerations—ICOMPRESS</u>	I-1
<u>IF/THEN/ELSE Clause</u>	I-1
<u>Considerations—IF/THEN/ELSE</u>	I-2
<u>Examples—IF/THEN/ELSE</u>	I-2
<u>IN Predicate</u>	I-3
<u>Considerations—IN</u>	I-3
<u>Examples—IN</u>	I-4
<u>INCLUDE SQLCA Directive</u>	I-4
<u>Consideration—INCLUDE SQLCA</u>	I-4
<u>Example—INCLUDE SQLCA</u>	I-4
<u>INCLUDE SQLDA Directive</u>	I-4
<u>Consideration—INCLUDE SQLDA</u>	I-6
<u>Example—INCLUDE SQLDA</u>	I-6
<u>INCLUDE SQLSA Directive</u>	I-6
<u>Consideration—INCLUDE SQLSA</u>	I-6
<u>Example—INCLUDE SQLSA</u>	I-6
<u>INCLUDE STRUCTURES Directive</u>	I-7
<u>Considerations—INCLUDE STRUCTURES</u>	I-8
<u>Examples—INCLUDE STRUCTURES</u>	I-8

I. (continued)

Index Keys	I-9
INDEXES Table	I-10
Indicator Variables and Indicator Parameters	I-11
INFO DEFINE Command	I-12
Consideration—INFO DEFINE	I-12
Examples—INFO DEFINE	I-12
INITIALIZE SQL Command	I-13
Considerations—INITIALIZE SQL	I-13
Example—INITIALIZE SQL	I-13
INSERT Statement	I-14
Considerations—INSERT	I-16
Examples—INSERT	I-18
INTERVAL Data Type	I-19
Considerations—INTERVAL Data Type	I-20
Example—INTERVAL Data Type	I-21
INTERVAL Literals	I-22
Example—Interval Literals	I-24
INVOKE Directive and Command	I-25
Considerations—INVOKE	I-28
Examples—INVOKE	I-28
ISLACK File Attribute	I-29
Considerations—ISLACK	I-29

J.

Joins	J-1
Examples - Joins	J-1
JULIANTIMESTAMP Function	J-4
Example—JULIANTIMESTAMP	J-4

K.

Keys	K-1
KEYS Table	K-1

L.

LEFT_MARGIN Option	L-1
Example—LEFT_MARGIN	L-1
LIKE Predicate	L-2
Considerations—LIKE	L-2
Examples—LIKE	L-5

L. (continued)

- [Limits](#) L-6
- [LINE_NUMBER Function](#) L-14
 - [Considerations—LINE_NUMBER](#) L-15
 - [Examples—LINE_NUMBER](#) L-15
- [LINE_SPACING Option](#) L-16
 - [Examples—LINE_SPACING](#) L-16
- [LIST Command](#) L-16
 - [Considerations—LIST](#) L-17
 - [Example—LIST](#) L-17
- [Literals](#) L-18
- [LOAD Command](#) L-18
 - [Considerations—LOAD](#) L-33
 - [Example—LOAD](#) L-45
- [LOCK TABLE Statement](#) L-45
 - [Considerations—LOCK TABLE](#) L-46
 - [Examples—LOCK TABLE](#) L-47
- [Locking](#) L-48
 - [Lock Duration](#) L-48
 - [Lock Mode](#) L-50
 - [Lock Holder](#) L-50
- [LOCKLENGTH File Attribute](#) L-51
 - [Consideration—LOCKLENGTH](#) L-51
 - [Example—LOCKLENGTH](#) L-51
- [LOG Command](#) L-52
 - [Example—LOG](#) L-52
- [LOGICAL_FOLDING Option](#) L-53
 - [Considerations—LOGICAL_FOLDING](#) L-53
 - [Example—LOGICAL_FOLDING](#) L-53

M.

- [MAX Function](#) M-1
 - [Considerations—MAX](#) M-1
 - [Example—MAX](#) M-2
- [MAXEXTENTS File Attribute](#) M-2
 - [Considerations—MAXEXTENTS](#) M-2
- [Message File](#) M-3
- [MIN Function](#) M-3
 - [Considerations—MIN](#) M-4

M. (continued)

Example—MIN	M-4
MODIFY CATALOG	M-4
MODIFY CATALOG with REPLACE NODENAME	M-4
Considerations—MODIFY CATALOG with REPLACE NODENAME	M-8
Examples—MODIFY CATALOG with REPLACE NODENAME	M-10
MODIFY CATALOG with REPLACE VOLUME	M-11
Considerations-MODIFY CATALOG with REPLACE VOLUME	M-15
Examples-MODIFY CATALOG with REPLACE VOLUME	M-18
MODIFY LABEL	M-19
MODIFY LABEL with REPLACE NODENUMBER	M-19
Considerations—MODIFY LABEL with REPLACE NODENUMBER	M-24
Examples—MODIFY LABEL with REPLACE NODENUMBER	M-27
MODIFY LABEL with REPLACE VOLUME	M-30
Considerations-MODIFY LABEL with REPLACE VOLUME	M-34
Examples-MODIFY LABEL with REPLACE VOLUME	M-37
MODIFY LABEL with REPLACE VOLUME and Partitioned Objects	M-38
MODIFY REGISTER Command	M-40
Considerations—MODIFY REGISTER	M-42
Examples—MODIFY REGISTER	M-42
Multibyte Character Sets	M-43
System Default National Character Set	M-43

N.

NAME Command	N-1
Consideration—NAME Command	N-1
Example—NAME Command	N-1
NAME Option	N-2
Consideration—NAME Option	N-2
Example—NAME Option	N-2
Name Resolution	N-2
Names	N-3
NEWLINE_CHAR Option	N-4
Consideration—NEWLINE_CHAR	N-4
Example—NEWLINE_CHAR	N-4
Nonaudited Tables	N-4
NOPURGEUNTIL File Attribute	N-5
Defaults	N-5
Example—NOPURGEUNTIL	N-5

N. (continued)

- [NULL Predicate](#) N-6
 - [Considerations—NULL](#) N-6
 - [Examples—NULL](#) N-6
- [Null Values](#) N-7
 - [Using Null Values Versus Default Values](#) N-7
 - [Defining Columns That Allow or Prohibit Nulls](#) N-8
 - [Determining Whether a Column Allows Nulls](#) N-8
 - [Specifying Null Values in Host Programs](#) N-9
 - [DISTINCT, GROUP BY, and ORDER BY With Null Values](#) N-9
 - [Null Values and Expression Evaluation](#) N-10
- [NULL_DISPLAY Option](#) N-10
 - [Example—NULL_DISPLAY](#) N-10
- [Numeric Data Types](#) N-11
 - [Considerations—Numeric Data Types](#) N-12
- [Numeric Literals](#) N-13
 - [Example—Numeric Literals](#) N-13

O.

- [OBEY Command](#) O-1
 - [Considerations—OBEY](#) O-1
 - [Examples—OBEY](#) O-3
- [OCTET_LENGTH Function](#) O-4
 - [Considerations—OCTET LENGTH Function](#) O-4
 - [Examples—OCTET LENGTH Function](#) O-4
- [OPEN Statement](#) O-5
 - [Considerations—OPEN](#) O-5
 - [Example—OPEN](#) O-6
- [OSS NAMES](#) O-6
- [OUT Command](#) O-7
 - [Consideration—OUT](#) O-7
 - [Example—OUT](#) O-8
- [OUT_REPORT COMMAND](#) O-8
 - [Considerations—OUT_REPORT](#) O-9
 - [Examples—OUT_REPORT](#) O-9
- [OVERFLOW_CHAR OPTION](#) O-10
 - [Consideration—OVERFLOW_CHAR](#) O-10
 - [Example—OVERFLOW_CHAR](#) O-10
- [OWNER FILE ATTRIBUTE](#) O-10

P.

PAGE_COUNT Option	P-1
Consideration—PAGE_COUNT	P-1
Example—PAGE_COUNT	P-1
PAGE FOOTING Command	P-1
Considerations—PAGE FOOTING	P-2
Example—PAGE FOOTING	P-2
PAGE_LENGTH Option	P-2
Considerations—PAGE_LENGTH	P-3
Example—PAGE_LENGTH	P-3
PAGE_NUMBER Function	P-3
Considerations—PAGE_NUMBER	P-3
Example—PAGE_NUMBER	P-3
PAGE TITLE Command	P-4
Considerations—PAGE TITLE	P-4
Example—PAGE TITLE	P-4
Parallel Index Loading	P-5
Default Configuration for Parallel Index Loading	P-5
Specifying Configuration for Parallel Index Loading	P-6
Consideration—Parallel Index Loading	P-10
Sample Configuration File	P-11
Parameters	P-11
Considerations—Parameters	P-13
Example—Parameters	P-15
PARTITION Clause	P-16
Consideration—PARTITION	P-18
Example—PARTITION	P-18
Partitions	P-19
PARTNS Table	P-20
PERUSE Command	P-21
Example—PERUSE	P-21
Plans	P-22
POSITION Function	P-23
Considerations—POSITION Function	P-23
Examples—POSITION Function	P-24
Predicates	P-24
PREPARE Statement	P-25
Considerations—PREPARE	P-26
Examples—PREPARE	P-26

P. (continued)

- [Primary Keys](#) P-27
- [Print Item](#) P-28
- [PROGID File Attribute](#) P-28
- [Program Invalidation](#) P-28
 - [Operations That Invalidate a Program](#) P-29
 - [Preventing Program Invalidation Caused by DDL Operations](#) P-29
- [PROGRAMS Table](#) P-31
- [Protection View](#) P-32
- [PURGE Command](#) P-33
 - [Considerations—PURGE](#) P-34
 - [Examples—PURGE](#) P-36
- [PURGEDATA Command](#) P-36
 - [Considerations—PURGEDATA](#) P-38
 - [Examples—PURGEDATA](#) P-39

Q.

- [Qualified Fileset List](#) Q-1
 - [Examples—Qualified Fileset List](#) Q-5
- [Quantified Predicate](#) Q-6
 - [Considerations—Quantified Predicate](#) Q-6
 - [Examples—Quantified Predicate](#) Q-6

R.

- [RECLength File Attribute](#) R-1
 - [Considerations—RECLength](#) R-1
- [RELEASE Statement](#) R-1
- [REPORT FOOTING Command](#) R-2
 - [Considerations—REPORT FOOTING](#) R-2
 - [Example—REPORT FOOTING](#) R-2
- [REPORT Option](#) R-3
 - [Considerations—REPORT Option](#) R-3
 - [Examples—REPORT Option](#) R-3
- [REPORT TITLE Command](#) R-6
 - [Considerations—REPORT TITLE](#) R-6
 - [Examples—REPORT TITLE](#) R-6
- [Report Writer](#) R-7
- [Reserved Words](#) R-11
- [RESET DEFINE Command](#) R-12

R. (continued)

- [Considerations—RESET DEFINE](#) R-12
- [Example—RESET DEFINE](#) R-13
- [RESET LAYOUT Command](#) R-13
- [Example—RESET LAYOUT](#) R-14
- [RESET PARAM Command](#) R-14
- [Considerations—RESET PARAM](#) R-14
- [Examples—RESET PARAM](#) R-14
- [RESET PREPARED Command](#) R-16
- [Example—RESET PREPARED](#) R-16
- [RESET REPORT Command](#) R-16
- [Consideration—RESET REPORT](#) R-18
- [Examples—RESET REPORT](#) R-18
- [RESET SESSION Command](#) R-19
- [Example—RESET SESSION](#) R-19
- [RESET STYLE Command](#) R-19
- [Example—RESET STYLE](#) R-20
- [RESETBROKEN File Attribute](#) R-20
- [RIGHT_MARGIN Option](#) R-20
- [Consideration—RIGHT_MARGIN](#) R-21
- [Example—RIGHT_MARGIN](#) R-21
- [ROLLBACK WORK Statement](#) R-21
- [Considerations—ROLLBACK WORK](#) R-22
- [Example—ROLLBACK WORK](#) R-22
- [ROWCOUNT Option](#) R-23
- [Example—ROWCOUNT](#) R-23

S.

- [Sample Database](#) S-1
- [SAVE Command](#) S-2
- [Example—SAVE](#) S-4
- [Search Conditions](#) S-5
- [Considerations—Search Conditions](#) S-6
- [Examples—Search Conditions](#) S-6
- [SECURE Command](#) S-7
- [Considerations—SECURE Command](#) S-9
- [Examples—SECURE Command](#) S-10
- [SECURE File Attribute](#) S-11
- [Considerations—SECURE File Attribute](#) S-11

S. (continued)

Examples—SECURE File Attribute	S-11
Security	S-11
User IDs	S-12
Group Manager and Super ID	S-12
Process Access IDs	S-13
File Ownership	S-14
Security Strings	S-14
Authorization Requirements for SQL Statements	S-15
SELECT Statement	S-18
Considerations—SELECT	S-24
Considerations for UNION	S-25
Characteristics of UNION Columns	S-25
ORDER BY clause and UNION operator	S-26
GROUP BY Clause, HAVING Clause, and the UNION Operator	S-27
UNION ALL and Associativity	S-27
Examples—SELECT	S-27
SERIALWRITES File Attribute	S-32
Considerations—SERIALWRITES	S-32
SET DEFINE Command	S-32
Considerations—SET DEFINE	S-33
Example—SET DEFINE	S-33
SET DEFMODE Command	S-34
Example—SET DEFMODE	S-34
SET LAYOUT Command	S-34
Example—SET LAYOUT	S-35
SET PARAM Command	S-35
Considerations—SET PARAM	S-36
Examples—SET PARAM	S-37
SET SESSION Command	S-39
Considerations—SET SESSION	S-42
Examples—SET SESSION	S-44
SET STYLE Command	S-45
Consideration—SET STYLE	S-46
Example—SET STYLE	S-46
SETSCALE Function	S-46
Consideration—SETSCALE	S-46
Examples—SETSCALE	S-47
Shorthand View	S-48

S. (continued)

<u>SHOW CONTROL Command</u>	S-48
<u>Example—SHOW CONTROL</u>	S-48
<u>SHOW DEFINE Command</u>	S-48
<u>Consideration—SHOW DEFINE</u>	S-49
<u>Examples—SHOW DEFINE</u>	S-49
<u>SHOW DEFMODE Command</u>	S-49
<u>Example—SHOW DEFMODE</u>	S-49
<u>SHOW LAYOUT Command</u>	S-50
<u>Example—SHOW LAYOUT</u>	S-50
<u>SHOW PARAM Command</u>	S-50
<u>Examples—SHOW PARAM</u>	S-51
<u>SHOW PREPARED Command</u>	S-51
<u>Example—SHOW PREPARED</u>	S-51
<u>SHOW REPORT Command</u>	S-52
<u>Example—SHOW REPORT</u>	S-52
<u>SHOW SESSION Command</u>	S-53
<u>Example—SHOW SESSION</u>	S-53
<u>SHOW STYLE Command</u>	S-54
<u>Example—SHOW STYLE</u>	S-54
<u>Similarity Checks</u>	S-54
<u>General Rules for Similarity</u>	S-55
<u>Similarity Between Protection Views</u>	S-55
<u>Similarity Between Tables</u>	S-55
<u>Similarity Between Collations</u>	S-57
<u>Displaying the Similarity Check Attribute</u>	S-57
<u>SLACK File Attribute</u>	S-57
<u>Purpose of SLACK</u>	S-57
<u>SPACE Option</u>	S-58
<u>Considerations—SPACE</u>	S-58
<u>Examples—SPACE</u>	S-58
<u>SQL Directive</u>	S-59
<u>SQL Identifiers</u>	S-59
<u>SQLCI</u>	S-60
<u>An SQLCI Session</u>	S-60
<u>The SQLCI Command</u>	S-61
<u>Considerations—SQLCI</u>	S-62
<u>Example—SQLCI</u>	S-63
<u>SQLCI Commands</u>	S-63

S. (continued)

SQLCODE	S-67
SQLCOMP Command	S-67
Standards Conformance	S-67
Exceptions to Conformance With Entry Level SQL 1992	S-67
SQL/MP Features From Intermediate Level SQL 1992	S-70
SQL/MP Features From Full Level SQL 1992	S-71
SQL/MP Extensions to SQL 1992	S-71
Statements	S-73
Static SQL	S-77
Statistics	S-78
Storage Management Foundation (SMF)	S-78
Considerations—SMF	S-79
String Functions	S-80
String Literals	S-80
Considerations—String Literals	S-81
Examples—String Literals	S-82
Subqueries	S-82
Considerations—Subqueries	S-83
SUBSTRING Function	S-84
Considerations—SUBSTRING Function	S-85
Examples—SUBSTRING Function	S-85
SUBTOTAL Command	S-86
Considerations—SUBTOTAL	S-86
Examples—SUBTOTAL	S-87
SUBTOTAL_LABEL Option	S-88
Considerations—SUBTOTAL_LABEL	S-88
Example—SUBTOTAL_LABEL	S-89
SUM Function	S-89
Considerations—SUM	S-89
Example—SUM	S-90
Super ID	S-90
Syskeys	S-90
System Catalog	S-92
SYSTEM Command	S-93
Considerations—SYSTEM	S-93
Example—SYSTEM	S-93
System DEFINES	S-94

T.

TABLECODE File Attribute	T-1
Tables	T-1
TABLES Table	T-2
TEDIT Command	T-3
Example—TEDIT	T-3
Temporary Tables	T-4
TIME_FORMAT Option	T-4
Example—TIME_FORMAT	T-4
TIME Data Type	T-5
Example—TIME Data Type	T-5
TIMESTAMP Data Type	T-5
Example—TIMESTAMP Data Type	T-5
TMF Transactions	T-6
Transaction Control Statements	T-6
User-Defined and System-Defined Transactions	T-6
Rules for DDL and DML Statements	T-7
Rules for SQLCI	T-8
Rules for Host Programs	T-8
TOTAL Command	T-9
Considerations—TOTAL	T-9
Examples—TOTAL	T-10
TRANSIDS Table	T-11
TRIM Function	T-11
Consideration—TRIM Function	T-12
Examples—TRIM Function	T-12

U.

UNDERLINE_CHAR Option	U-1
Example—UNDERLINE_CHAR	U-1
UNLOCK TABLE Statement	U-1
Considerations—UNLOCK TABLE	U-1
Examples—UNLOCK TABLE	U-2
UPDATE Statement	U-3
Considerations—UPDATE	U-4
Examples—UPDATE	U-6
UPDATE STATISTICS Statement	U-7
Considerations—UPDATE STATISTICS	U-8
Examples—UPDATE STATISTICS	U-10

U. (continued)

- [UPGRADE CATALOG Command](#) U-11
 - [Considerations—UPGRADE CATALOG](#) U-12
 - [Examples—UPGRADE CATALOG](#) U-12
- [UPGRADE SYSTEM CATALOG Command](#) U-13
 - [Considerations—UPGRADE SYSTEM CATALOG](#) U-13
 - [Example—UPGRADE SYSTEM CATALOG](#) U-13
- [UPSHIFT Function](#) U-14
 - [Considerations—UPSHIFT](#) U-14
 - [Examples—UPSHIFT](#) U-14
- [USAGES Table](#) U-15
- [User-Defined Keys](#) U-16
- [Utilities](#) U-16

V.

- [VARCHAR_WIDTH Option](#) V-1
 - [Consideration—VARCHAR_WIDTH](#) V-1
 - [Example—VARCHAR_WIDTH](#) V-1
- [VERIFIEDWRITES File Attribute](#) V-1
 - [Consideration—VERIFIEDWRITES](#) V-2
- [VERIFY Command](#) V-2
 - [Considerations—VERIFY](#) V-3
 - [Examples-VERIFY](#) V-5
- [Versions](#) V-6
 - [SQL/MP Component Versions](#) V-7
 - [Catalog Versions](#) V-7
 - [Object Versions](#) V-7
 - [Program Versions](#) V-8
 - [Host language compiler versions](#) V-8
- [VERSIONS Table](#) V-9
- [Views](#) V-9
- [VIEWS Table](#) V-10
- [VOLUME Command](#) V-11
 - [Considerations—VOLUME](#) V-11
 - [Examples—VOLUME](#) V-11

W.

- [WHENEVER DIRECTIVE](#) W-1
 - [Consideration—WHENEVER Directive](#) W-2
- [WHERE CLAUSE](#) W-2
- [WINDOW OPTION](#) W-2
 - [Consideration—WINDOW](#) W-3
 - [Examples—WINDOW](#) W-3
- [WITH SHARED ACCESS OPTION](#) W-4
 - [Considerations—WITH SHARED ACCESS](#) W-5
 - [Example—WITH SHARED ACCESS](#) W-8

Z.

- [! Command](#) Z-1
 - [Examples—!](#) Z-1
- [= AUDSERV_XSWAP *node* DEFINE](#) Z-2
 - [Consideration—= AUDSERV_XSWAP *node*](#) Z-2
 - [Example—= AUDSERV_XSWAP *node*](#) Z-2
- [= DEFAULTS DEFINE](#) Z-3
 - [Considerations—= DEFAULTS](#) Z-3
 - [Examples—= DEFAULTS](#) Z-4
- [= SORT_DEFAULTS DEFINE](#) Z-4
 - [Considerations—= SORT_DEFAULTS](#) Z-5
 - [Example—= SORT_DEFAULTS](#) Z-6
- [= SQL_CAT_HEAP_LIMIT DEFINE](#) Z-6
 - [Considerations—= SQL_CAT_HEAP_LIMIT](#) Z-6
 - [Examples—= SQL_CAT_HEAP_LIMIT](#) Z-6
- [= SQL_CMP_CPUS *node* DEFINE](#) Z-7
 - [Considerations—= SQL_CMP_CPUS *node*](#) Z-7
- [= SQL_CMP_DOUBLE_SBB_OFF DEFINE](#) Z-8
 - [Consideration—= SQL_CMP_DOUBLE_SBB_OFF](#) Z-9
- [= SQL_CMP_DOUBLE_SBB_ON DEFINE](#) Z-9
 - [Considerations—= SQL_CMP_DOUBLE_SBB_ON](#) Z-9
- [= SQL_CMP_EQ_LIMIT DEFINE](#) Z-10
 - [Consideration—= SQL_CMP_EQ_LIMIT](#) Z-10
 - [Examples—= SQL_CMP_EQ_LIMIT](#) Z-10
- [= SQL_CMP_EVENT DEFINE](#) Z-11
 - [Format of SQL Compiler Event Messages](#) Z-11
 - [Considerations—= SQL_CMP_EVENT](#) Z-12
 - [Examples—= SQL_CMP_EVENT](#) Z-12

Z. (continued)

- [= SQL_CMP_EVENT_NO0 DEFINE](#) Z-12
 - [Default Event Messages](#) Z-13
 - [Consideration—= SQL_CMP_EVENT_NO0](#) Z-13
 - [Example—= SQL_CMP_EVENT_NO0](#) Z-13
- [= SQL_CMP_NO_KS_MJOIN DEFINE](#) Z-14
 - [Examples—= SQL_CMP_NO_KS_MJOIN](#) Z-14
- [= SQL_cmp_node DEFINE](#) Z-14
 - [Consideration—= SQL_cmp_node](#) Z-15
 - [Example—= SQL_cmp_node](#) Z-15
- [= SQL_EXE_DOUBLE_SHUTOFF DEFINE](#) Z-15
 - [Considerations—= SQL_EXE_DOUBLE_SHUTOFF](#) Z-15
- [= SQL_EXE_ESPS_CK_CM0N DEFINE](#) Z-16
 - [Consideration—= SQL_EXE_ESPS_CK_CM0N](#) Z-16
 - [Example—= SQL_EXE_ESPS_CK_CM0N](#) Z-16
- [= SQL_EXE_USE_SWAPVOL DEFINE](#) Z-17
 - [Considerations—= SQL_EXE_USE_SWAPVOL](#) Z-17
 - [Examples—= SQL_EXE_USE_SWAPVOL](#) Z-17
- [= SQL_MSG_node DEFINE](#) Z-18
 - [Considerations—= SQL_MSG_node](#) Z-18
 - [Examples—= SQL_MSG_node](#) Z-19
- [= SQL_RECGEN_node DEFINE](#) Z-20
 - [Example—= SQL_RECGEN_node](#) Z-20
- [= SQL_TM_node_vol DEFINE](#) Z-20
 - [Considerations—= SQL_TM_node_vol](#) Z-21
 - [Examples—= SQL_TM_node_vol](#) Z-21

Index

Examples

- [Example R-1.](#) [Report Example](#) R-4

Tables

- [Table A-1.](#) [Transaction Concurrency Depending on Access Option](#) A-3
- [Table A-2.](#) [Display Descriptors for Character Items](#) A-61
- [Table A-3.](#) [Display Descriptors for Numeric Items](#) A-61
- [Table A-4.](#) [Scale-Sign Descriptors](#) A-62
- [Table A-5.](#) [Modifiers](#) A-62
- [Table A-6.](#) [ASCII Character Set](#) A-70
- [Table B-1.](#) [The BASETABS Table](#) B-1

Tables (continued)

Table C-1.	Catalog Tables and Indexes	C-8
Table C-2.	The SQL.CATALOGS Table	C-11
Table C-3.	The COLUMNS Table	C-46
Table C-4.	The COMMENTS Table	C-51
Table C-5.	Summary of Concurrent DDL and DML Operations	C-65
Table C-6.	ALTER Operation Effects on Timestamps	C-67
Table C-7.	Limits on Concurrent Utility and DML Operations	C-68
Table C-8.	The CONSTRNT Table	C-70
Table C-9.	The CPRLSRCE Table	C-131
Table C-10.	The CPRULES Table	C-131
Table D-1.	Attributes of DEFINES (by Class)	D-34
Table E-1.	EXPLAIN Plan Header Lines	E-16
Table F-1.	FILEINFO BRIEF Display Information Line	F-11
Table F-2.	The FILES Table	F-27
Table F-3.	FUP Commands and SQL Objects	F-34
Table I-1.	The INDEXES Table	I-10
Table K-1.	The KEYS Table	K-1
Table L-1.	Lock Release Summary	L-49
Table N-1.	Numeric Data Types in SQL—Binary Types	N-11
Table N-2.	Numeric Data Types in SQL—Floating Point Types	N-12
Table N-3.	Numeric Data Types in SQL—Decimal Types	N-12
Table P-1.	The PARTNS Table	P-20
Table P-2.	The PROGRAMS Table	P-31
Table R-1.	SQLCI Commands Used to Write Reports	R-8
Table R-2.	Style and Layout Options for Reports	R-10
Table R-3.	Report Writer Clauses	R-10
Table R-4.	Report Writer Functions	R-11
Table R-5.	Reserved Words	R-11
Table S-1.	Authorization Requirements for SQL Statements	S-15
Table S-2.	Summary of SQLCI Commands	S-64
Table S-3.	Summary of SQL Statements	S-73
Table S-4.	Summary of SQL Statements	S-77
Table T-1.	The TABLES Table	T-2
Table T-2.	TRANSIDS Table	T-11
Table U-1.	The USAGES Table	U-15
Table V-1.	The VERSIONS Table	V-9
Table V-2.	The VIEWS Table	V-10

What's New in This Manual

Manual Information

Abstract

This manual describes HP NonStop™ SQL/MP, the HP relational database management system that uses SQL to describe and manipulate data in a NonStop SQL/MP database. The manual includes information about SQLCI, the conversational interface to NonStop SQL/MP.

Product Version

NonStop SQL/MP G07 and H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, G06.27 and all subsequent G-series RVUs, and D46.00 and all subsequent D-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
523352-013	April 2013

Document History

Part Number	Product Version	Published
523352-009	NonStop SQL/MP G07 and H01	July 2008
523352-010	NonStop SQL/MP G07 and H01	August 2008
523352-011	NonStop SQL/MP G07 and H01	November 2008
523352-012	NonStop SQL/MP G07 and H01	August 2010
523352-013	NonStop SQL/MP G07 and H01	April 2013

New and Changed Information

Changes to the 523352-013 manual:

- Updated the section Examples—CREATE INDEX on page [C-151](#).
- Updated the section [MAXEXTENTS File Attribute](#) on page M-2.

Changes to the 523352-012 manual:

- Updated [Considerations—PURGEDATA](#) on page P-38.
- Update header size for Format 2 key-sequenced tables on page [L-13](#).
- Added consideration for LOAD command on page [L-45](#).
- Updated [Lock Holder](#) on page L-50.
- Updated SYSKEY information on [S-91](#).

Changes to the H06.16/J06.05 Manual

- Updated the [CREATE CATALOG Statement](#) on page C-132.
- Updated the maximum number of expression allowed in the expression list for the [IN Predicate](#) on page I-3.

Changes to the H06.15/J06.04 Manual

- Supported release statements have been updated to include J-series RVUs.
- Updated information about the [MODIFY CATALOG](#) replace specifications as:
 - [MODIFY CATALOG with REPLACE NODENAME](#) on page M-5.
 - [MODIFY CATALOG with REPLACE VOLUME](#) on page M-12.
- Updated information about the [MODIFY LABEL](#) replace specifications as:
 - [MODIFY LABEL with REPLACE NODENUMBER](#) on page M-20.
 - [MODIFY LABEL with REPLACE VOLUME](#) on page M-31.
- Updated the index to include command entries.

Changes to the G06.32 Manual

- Updated the considerations for the CAST function under [Valid Considerations—CAST](#) on page C-5.
- Updated the description of AUDITONLY option under [COMMIT WORK Statement](#) on page C-56.

- Modified the description for setting the PROBABALISTIC option under [UPDATE STATISTICS Statement](#) on page U-7.

Changes to the G06.30 Manual

- Modified the renaming tables description under Considerations—ALTER TABLE on page [A-38](#).
- Modified the directives under Considerations—CONTROL TABLE on pages [C-85](#) and [C-86](#).
- Updated the description for setting the ERROR ABORT parameter on page [S-40](#) under SET SESSION Command.
- Added the considerations for interactive SQLCI under Considerations—SET SESSION on page [S-43](#).

About This Manual

This manual is the main reference text for NonStop SQL/MP, the HP relational database management system based on SQL. This manual includes reference information about the programmatic and conversational interfaces to NonStop SQL/MP, in addition to information about utilities used to install and maintain a NonStop SQL/MP database.

This manual is structured as an encyclopedia-style reference text. Entries that describe statements, commands, and conceptual information are intermixed in an alphabetic order for easy look-up. If you know what you are looking for, you can probably turn directly to a specific entry, but there is no introductory or overview information.

If you are new to NonStop SQL/MP, read the *Introduction to NonStop SQL/MP* before you begin to use this manual. If you have read the *Introduction to NonStop SQL/MP* and you are browsing this manual, begin with the table of contents or one of these entries:

- Statements
- SQLCI
- SQLCI Commands

The text of this manual is also available online through SQLCI. For information about displaying manual entries through SQLCI, see [HELP Command](#) on page H-2.

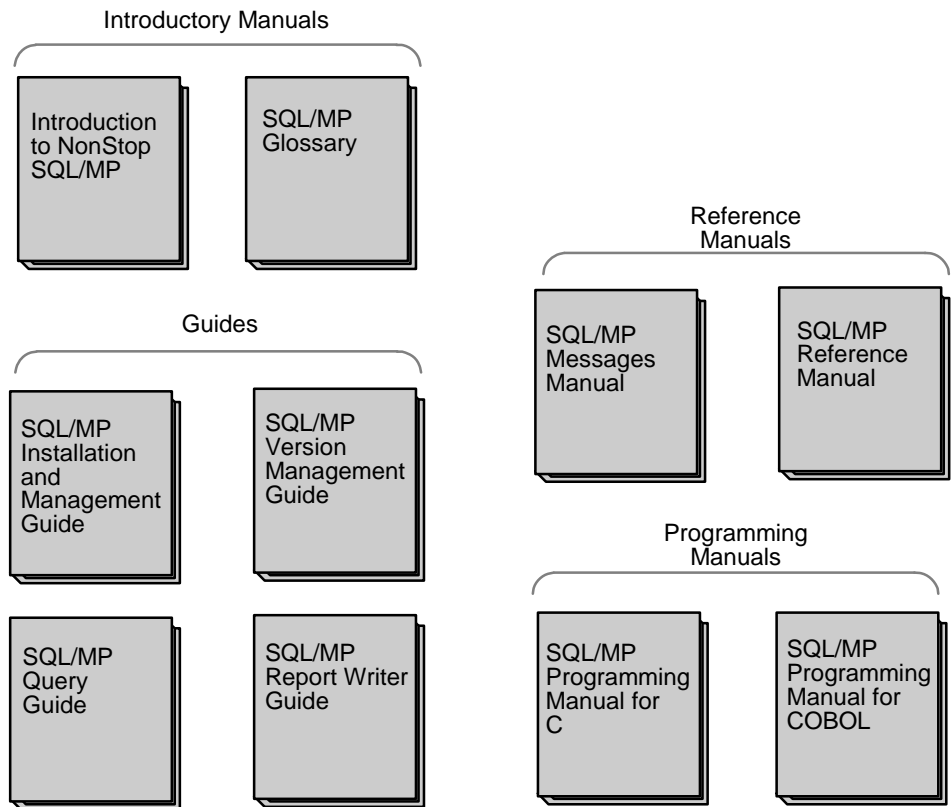
Related Manuals

You might want to use some books other than the SQL/MP library set with this reference manual. The complete library includes:

Manual Title	Description
<i>Introduction to NonStop SQL/MP</i>	Provides an overview of the NonStop SQL/MP relational database management system.
<i>SQL/MP Glossary</i>	Provides definitions of terms used in the SQL/MP documentation.
<i>SQL/MP Installation and Management Guide</i>	Explains how to plan, install, create, and manage a NonStop SQL database.
<i>SQL/MP Messages Manual</i>	Describes messages returned by NonStop SQL/MP.
<i>SQL/MP Query Guide</i>	Describes how to write SQL/MP queries, and how to optimize queries for enhanced performance.
<i>SQL/MP Report Writer Guide</i>	Describes how to use SQLCI report writer commands to design and produce reports.

Manual Title	Description
<i>SQL/MP Version Management Guide</i>	Describes the rules that govern the version management for the NonStop SQL/MP relational database management system.
<i>SQL/MP Programming Manual for C and SQL/MP Programming Manual for COBOL</i>	Describe the NonStop SQL/MP programmatic interfaces for C and COBOL, respectively.
<i>SQL Programming Manual for Pascal and SQL Programming Manual for TAL</i>	Describe the NonStop SQL/MP programmatic interfaces for Pascal and TAL, respectively.

This figure shows the manuals in the SQL/MP library:



VST001.vsd

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:

To view the related manuals, click on [Related Manuals](#).

General Syntax Notation

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

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

```
MAXATTACH
```

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

```
file-name
```

[] 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:

```
LIGHTS [ ON                ]
        [ OFF              ]
        [ SMOOTH [ num ] ]

K [ X | D ] address-1
```

{ } Braces. A group of items enclosed in braces is a list from which you 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 }
```

Required Choice { | }. Required choice indicators enclose multiple required syntax items. A vertically aligned group of items enclosed in required choice indicators represents a list of selections from which you must choose one or more, in any order, but cannot repeat a selection.

Optional Choice [|]. Optional choice indicators enclose multiple optional syntax items. A vertically aligned group of items enclosed in optional choice indicators represents a list of selections from which you can choose one or more, in any order, but cannot repeat a selection.

| 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-1 [ , new-value ]...
```

```
[ - ] { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }...
```

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

```
"s-char..."
```

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

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

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

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

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

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

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

If there is no space between two items, spaces are not permitted. In this example, 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 / ] CONTROLLER  
      [ , attribute-spec ]...
```

Change Bar Notation

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

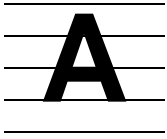
The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).

The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

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

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



Access Options

Access options on DDL or DML statements determine the locking or access mode that SQL uses when executing the statements. Access options affect the consistency of the data your application views and the degree of concurrency your program has with other programs that use the same data.

Access Options on DML Statements

The DML statements SELECT, INSERT, UPDATE, and DELETE include access options that control lock duration and have row-level granularity. The possible settings for access options on DML statements are:

BROWSE	Does not lock data. Reads locked data that might be inconsistent. Used for browse access or when a table is locked. Minimum consistency, maximum concurrency.
STABLE	Locks all data accessed but releases locks on unmodified rows without waiting for the end of the transaction. The default for read operations.
REPEATABLE	Locks all data accessed until the end of the transaction. Maximum consistency, minimum concurrency. The default for update operations.

BROWSE access is not available on DML statements that modify the database.

STABLE access and REPEATABLE access are available on DML statements that modify the database. STABLE access and REPEATABLE access are discussed in detail in the coming subsections.

STABLE Access Option on DML Statements

STABLE locks all data accessed through the DML statement but releases locks on unmodified data as soon as possible. STABLE access locks modified data in audited tables until the end of the transaction.

In host programs that use cursors, STABLE locks an unmodified row only when the row is in the current position and releases the lock at the next FETCH on the cursor. CLOSE cursor releases the lock from the last FETCH.

For modified rows in audited tables, STABLE access uses exclusive locks held by the HP NonStop Transaction Management Facility (TMF) transaction that are released only when the entire transaction ends.

For nonaudited tables, all locks are held by the process (not the transaction) and STABLE access releases locks when the statement finishes.

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

REPEATABLE Access Option on DML Statements

The REPEATABLE access option locks all data accessed through the DML statement and holds the locks on data in audited tables until the end of the transaction.

For audited tables, REPEATABLE access uses shared locks for unmodified rows and exclusive locks for modified rows—but all locks are held by the TMF transaction and not released until the transaction ends. For audited tables, REPEATABLE access prevents other users from inserting or modifying rows in the range of rows examined by the DML statement.

For nonaudited tables, locks are held by the executing process, not the transaction, but ending a transaction normally releases locks on both audited and nonaudited tables. To hold locks on nonaudited data across multiple transactions, specify the AUDITONLY option on COMMIT WORK, ROLLBACK WORK, and FREE RESOURCES statements within the transaction.

For nonaudited tables outside of transactions in host programs, locks acquired with REPEATABLE access remain in effect until the program releases them with UNLOCK TABLE or FREE RESOURCES. In programs that use cursors, a lock on a row takes effect when the FETCH for the row executes.

For nonaudited tables, if sequential block buffering (either RSBB or VSBB) is used to access a nonaudited table, a table lock is always acquired regardless of the access option (stable or repeatable) or the CONTROL TABLE TABLELOCK option specification.

Within SQLCI-defined transactions when AUTOWORK is ON without the AUDITONLY option, REPEATABLE access locks nonaudited tables only until SQLCI commits or rolls back the transaction. If AUTOWORK is ON with the AUDITONLY option, REPEATABLE access locks nonaudited tables until you release them.

REPEATABLE does not provide full, repeatable read-protection for nonaudited tables within SQLCI or a host program. REPEATABLE does not prevent other users from inserting rows in the originally selected range of rows. Use the LOCK TABLE statement if you need such protection.

Access Options on DDL Statements

The only access option available on DDL statements is the WITH SHARED ACCESS option, which allows users to control whether or not concurrent DML statements can acquire write access to the SQL objects being changed by the DDL operation. Only a few DDL statements currently include the WITH SHARED ACCESS option.

DDL operations performed with the WITH SHARED ACCESS option allow DML operations by other processes to acquire read and write access to objects being changed during all but the final phase of the DDL operation. Most DDL operations

performed without the WITH SHARED ACCESS option allow read-only access for concurrent DML operations during most of the DDL operation.

For more information, see [WITH SHARED ACCESS OPTION](#) on page W-4. See the entry for the specific DDL statement you plan to use to determine if that statement allows the WITH SHARED ACCESS option.

Summary: Effect of Access Options on Concurrency

Concurrency is access to the same data by two or more processes at the same time. The degree of concurrency available (that is, whether a process that requests access to data already being accessed is given access or placed in a wait queue) depends on the purpose of the access (to read data or to update data), on the access option, and on whether SQL uses VSBB for the access.

For more information, see [Concurrency](#) on page C-65.

[Table A-1](#) indicates how access options affect concurrency by listing the accesses allowed for one transaction (B) to rows currently being accessed by another transaction (A). All operations indicated in this table use default locking.

Table A-1. Transaction Concurrency Depending on Access Option

TRANSACTION B (Access Request)	Lock Type	TRANSACTION A (Data Currently Accessed)			
		STABLE		REPEATABLE	
		S	E	S	E
BROWSE					
SELECT or cursor	None	I	I	I	I
STABLE OR REPEATABLE					
SELECT exclusive	E	W	W	W	W
SELECT share	S	I	W	I	W
SELECT default	1	1	W	1	W
FETCH for update	E ²	I	W	W	W
FETCH no update	S ^{2, 3}	I	W	I	W
INSERT, UPDATE, or DELETE	E	W	W	W	W

E = Exclusive lock (the default for update operations)

S = Shared lock (the default for read operations)

I = Immediate access

W = Wait until the lock is released

For cursors, FETCH acquires locks on the fetched row.

¹ SQL determines lock mode. SHARE mode is in effect unless the program has updated the table or view, after which EXCLUSIVE mode is in effect.

² Records that do not satisfy a WHERE clause are examined and not returned; these records are always locked with a shared lock.

³ Records returned are locked EXCLUSIVE if a DELETE WHERE CURRENT statement has been issued or if FOR EXCLUSIVE was specified.

Because the previous table does not show lock duration, STABLE and REPEATABLE access options appear similar for Transaction B. For more information on lock duration, about modifying default locking, and about the effects of locking on concurrency, see [Locking](#) on page L-48.

ADD DEFINE Command

ADD DEFINE is an SQLCI command that creates DEFINES in the current SQLCI session. (ADD DEFINE is similar to the TACL command ADD DEFINE and the OSS command `add_define`.)

```
ADD DEFINE { define
            { ( define [ , define ] ... ) }
            [ , LIKE other-define ] [ , attr value ] ... ;
```

define

is a name for the new DEFINE; the name cannot be same as the name of an existing DEFINE.

A DEFINE name must begin with an equal sign (=) followed by a letter and can contain 2 to 24 characters, including alphanumeric characters, hyphens (-), underscores (_), and circumflexes (^). Uppercase and lowercase characters are considered equivalent in DEFINE names. Defines supplied by HP start with an equal sign followed by an underscore character (=_).

The new DEFINE has the attributes and values of the working attribute set modified by any *attr value* pairs you specify, unless you specify a LIKE clause. (The working attribute set is a set of default attribute values used when you create a new DEFINE and do not explicitly specify its attributes. For more information about the working attribute set, see [SET DEFINE Command](#) on page S-32, [RESET DEFINE Command](#) on page R-12, and [SHOW DEFINE Command](#) on page S-48.)

LIKE *other-define*

specifies an existing DEFINE on which to model the new DEFINE. The new DEFINE will be identical to the DEFINE you specify in the LIKE clause except for specific attributes you specify with *attr value* pairs.

If you use the LIKE clause, you cannot specify the CLASS attribute.

attr value

specifies an attribute and its value for the new DEFINE. The attribute value pairs are applied in the order you specify them.

If you specify the CLASS attribute, specify it first in the list of attributes. (CLASS MAP is the default, unless the working attribute set includes a different value.)

Setting the CLASS attribute establishes a new set of attributes for the DEFINE and sets each attribute associated with that CLASS to its initial value. If you specify CLASS after you specify other attributes, the values you specified for the previous attributes are erased.

For more information about DEFINE attributes, see [DEFINEs](#) on page D-27.

Considerations—ADD DEFINE

- You cannot use ADD DEFINE unless the DEFMODE setting is ON. The OSS command add_define automatically sets DEFMODE ON, but the TACL form of ADD DEFINE does not. Before issuing a TACL ADD DEFINE command, set DEFMODE ON.
- A DEFINE stays in effect until you change it, delete it, or exit the SQLCI session in which you created it.
- Attributes you specify in an ADD DEFINE command do not become part of the working attribute set. For information about changing the working attribute set, see [SET DEFINE Command](#) on page S-32.
- If the value of an attribute is a Guardian name or a subvolume name, the name is expanded immediately using the current default node, the volume, and the subvolume.

Examples—ADD DEFINE

- This example adds a DEFINE named =PCAT and uses it as a catalog name in a CREATE CATALOG statement:

```
SET DEFMODE ON;
ADD DEFINE =PCAT, CLASS CATALOG, SUBVOL $VOL2.INVENT;
CREATE TABLE ACCOUNT
  ( ACCT_NO NUMERIC (9) UNSIGNED, ACCT_NAME CHAR (50) )
  CATALOG =PCAT;
```

- This example assigns the name \SYS1.\$VOL2.SALES.SALESREP to the DEFINE named =SALES_REP_ACCOUNTS:

```
SET DEFMODE ON;
ADD DEFINE =SALES_REP_ACCOUNTS, CLASS MAP,
  FILE \SYS1.$VOL2.SALES.SALESREP;
```

- You can accomplish the same result shown in the previous example by setting the CLASS attribute in the working attribute set and adding a DEFINE that includes the FILE attribute. This method is especially useful when you want to create a series of DEFINES with the same CLASS, such as:

```
SET DEFINE CLASS MAP ;
ADD DEFINE =SALES_REP_ACCTS, FILE $VOL2.SALES.SALESREP ;
ADD DEFINE =CUSTOMER_TABLE, FILE $VOL1.SALES.CUSTOMER ;
ADD DEFINE =ORDERS_TABLE, FILE $VOL2.SALES.ORDERS ;
```

```
ADD DEFINE =ODETAIL_TABLE,      FILE $VOL3.SALES.ODETAIL ;
ADD DEFINE =PARTS_TABLE,        FILE $VOL4.SALES.PARTS ;
```

AGGREGATE Functions

NonStop SQL/MP provides these aggregate functions:

AVG Function	Computes the average of a set of numbers.
COUNT Function	Counts the number of rows that result from a query or the number of rows that contain a distinct value in a specific column.
MAX Function	Determines a maximum value.
MIN Function	Determines a minimum value.
SUM Function	Computes the sum of a set of numbers.

For more information, see the entry for a specific function.

Alias

An alias is a name assigned to a column in the select list of the SELECT command using the SQLCI report writer NAME command. You can use an alias to refer to the column in other parts of a report definition, such as a DETAIL command.

Use aliases to define abbreviations for long column names and to assign informative names to columns that consist of expressions.

An alias is not the same as a detail alias, which is a name assigned to a print item using the NAME clause of the DETAIL command. You can use a detail alias in report formatting commands such as TOTAL and SUBTOTAL but not in the DETAIL command itself.

ALLOCATE File Attribute

ALLOCATE is a Guardian file attribute that reserves disk space for a file, or frees disk space previously reserved for a file that does not contain data. ALLOCATE applies to key-sequenced, relative, and entry-sequenced tables and indexes.

Allocating disk space in advance ensures that space is available when needed and avoids processing errors caused by full or fragmented disks during normal allocation-on-demand.

```
{ ALLOCATE num-extents }
{ DEALLOCATE }
```

The default is ALLOCATE 0.

ALLOCATE num-extents

specifies the number of extents to allocate in advance. The number must be an integer between 1 and the current value of the MAXEXTENTS file attribute.

Depending on your file configuration, you might not be able to allocate the full number of MAXEXTENTS.

For ALTER TABLE or ALTER INDEX, ALLOCATE allocates new extents until the total of new and existing extents equals the specified number.

DEALLOCATE

frees all unused allocated extents (that is, all allocated extents beyond the extent that contains the end-of-file). DEALLOCATE is valid only for ALTER TABLE or ALTER INDEX.

Considerations—ALLOCATE

- ALLOCATE and DEALLOCATE apply to all partitions of the specified file unless you include a PARTONLY clause on the statement that specifies the file attribute. Use the PARTONLY clause if you want to specify different numbers of extents for different partitions:
 - In a CREATE TABLE statement that defines the primary partition and four secondary partitions, ALLOCATE 40 allocates 40 extents to each of the five partitions.
 - In an ALTER TABLE statement that specifies PARTONLY, ALLOCATE 40 allocates additional extents so that the specified partition has a total of 40 extents.
 - In an ALTER TABLE statement that does not specify PARTONLY, ALLOCATE 40 allocates additional extents to each partition whose total is less than 40 to make the number of extents equal to 40. ALLOCATE has no effect on partitions with 40 or more extents.

- ALLOCATE affects the number of extents, but not the size of extents. The EXTENT file attribute determines the extent size.

ALTER CATALOG Statement

ALTER CATALOG is a DDL statement that alters security attributes for an entire catalog.

```
ALTER CATALOG catalog { { [NO]CLEARONPURGE }
                        { NOPURGEUNTIL date
                        { OWNER group,user
                        { SECURE "rwep" }
```

catalog

is the name of the catalog to alter (or an equivalent DEFINE). If HP NonStop Storage Management Foundation (SMF) is installed on your node, *catalog* must be either virtual or a direct name.

The clauses set these security-related file attributes for the catalog:

CLEARONPURGE	Controls disk erasure when files are dropped
NOPURGEUNTIL	Sets date after which drop is allowed
OWNER	Specifies owner
SECURE	Sets Guardian security string

For more information, see [System Catalog](#) on page S-92 or an entry for a specific attribute.

This statement does not affect the system catalog CATALOGS table. You must alter that table with the ALTER TABLE statement.

Considerations—ALTER CATALOG

- To alter security attributes for a catalog, you must be a generalized owner of the catalog.

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 an ALTER CATALOG statement while another process is executing a DDL operation on an object in the catalog.

The specific error depends on the DDL operation involved and the phase of the operation during which the conflict occurs. For more information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

- Security information for a catalog is stored in catalog tables and file labels for the catalog. ALTER CATALOG changes the information in the catalog and the associated file labels on a disk.
- Changing ownership or security for a catalog can affect users of objects described in the catalog, so be careful when you narrow the set of users with read or write access. If you remove a user's authority to read or write to a catalog, the user cannot query, alter, or drop an object in the catalog (even if the user owns the object), or recompile a program that uses an object in the catalog.

SQL-compiling a program requires the authority to write to the PROGRAMS, USAGES, and TRANSIDS catalog tables in the catalog that contains the description of the program and to the USAGES and TRANSIDS catalog tables in any catalogs that contain descriptions of tables or views used by the program. Because of this requirement, you might want to secure these catalog tables independently from other catalog tables. You can use ALTER TABLE to set the security for the PROGRAMS, USAGES, and TRANSIDS tables.

Creating or dropping a catalog requires the authority to write to the system directory of catalogs in the SQL.CATALOGS table. Therefore, altering write authority for SQL.CATALOGS can prevent users from creating new catalogs or dropping existing catalogs. Because of this requirement, you might want to secure SQL.CATALOGS independently from the other tables in the system catalog. You can use ALTER TABLE to set the security for SQL.CATALOGS.

- Changing the OWNER attribute of a catalog affects the interpretation of the SECURE file attribute, because authorization is determined at run time using the current group and owner.

If another process is using a catalog when the owner changes, the process might not be able to reaccess the catalog after the change.

Example—ALTER CATALOG

This example makes user 201,43 the owner of the catalog named SALES, gives read and execute authority to all local and remote users, and gives write and purge authority to all users in group 201:

```
ALTER CATALOG SALES OWNER 201,43 SECURE "NUNU" ;
```

ALTER COLLATION Statement

ALTER COLLATION is a DDL statement that renames a collation or alters security attributes for a collation.

ALTER COLLATION <i>collation</i>	{	RENAME <i>new-name</i>		}
		OWNER <i>group,user</i>		}
		SECURE " <i>rwep</i> "		}

collation

is the name of the collation to alter (or an equivalent DEFINE). If SMF is installed on your node, *collation* must be either a virtual name or a direct name.

RENAME *new-name*

specifies a new Guardian name (or an equivalent DEFINE) for the object. SQL changes all references in the catalog to the new name.

OWNER or SECURE

specifies the owner and the security for the collation. For more information, see [OWNER FILE ATTRIBUTE](#) on page O-10, [Security](#) on page S-11, or [SECURE File Attribute](#) on page S-11.

Considerations—ALTER COLLATION

- ALTER COLLATION requires read and write authority for the collation and the catalog in which the collation is registered.

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 an ALTER COLLATION statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For more information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

- Restricting access to a collation effectively restricts access to objects that use the collation. This restriction is similar to restrictions for tables and views, but because collations are used differently from tables and views, you might want to use less restrictive security.

This scenario illustrates the type of problem that can occur if you alter the security for a collation to make it more restrictive:

1. User A creates a collation available to user B.
2. User B creates a table that uses the collation.
3. User A alters the security of the collation such that user B can no longer access it.

4. User B tries to compile SQL statements that reference user B's own table, but the compilation fails because user B does not have authority to access the collation.

Examples—ALTER COLLATION

- This example renames a collation:

```
ALTER COLLATION ORDER1 RENAME ORDERA;
```

- This example resecures a collation such that all users on the node can access it, but only the generalized owner can write to or purge it:

```
ALTER COLLATION TRN31 SECURE "A000";
```

ALTER DEFINE Command

ALTER DEFINE is an SQLCI command that changes the attributes of DEFINES in the current SQLCI session. (ALTER DEFINE is similar to the TACL command ALTER DEFINE.)

```
ALTER DEFINE define-list { , attr value
                           { , RESET reset-list } ... ;
```

define-list is:

```
{ define
  ( define [ , define ] ... )
  **
  =* }
```

reset-list is:

```
{ attr
  ( attr [ , attr ]... ) }
```

define

is the name of an existing DEFINE to alter.

** or =*

specifies all DEFINES.

attr value

specifies an attribute and its value for the DEFINE.

If you specify the CLASS attribute, specify it first in the list of attributes. Setting the CLASS attribute establishes a new set of attributes for the DEFINE and sets each attribute associated with that CLASS to its initial value. If you specify CLASS after

you specify other attributes, the values you specified for the previous attributes are erased. For more information about DEFINE attributes, see [DEFINES](#) on page D-27.

`RESET reset-list`

restores the value of each attribute listed in *reset-list* to its initial value.

You cannot reset a required attribute, so you cannot use this clause with CLASS MAP and CLASS CATALOG DEFINES.

Considerations—ALTER DEFINE

- When you end an SQLCI session, DEFINES that you inherited from another process (such as TACL) and modified within SQLCI revert to the values they had when you started SQLCI. Changes you make to inherited attributes from SQLCI apply only within SQLCI.
- ALTER DEFINE affects only existing DEFINES, not the working attribute set. (SET DEFINE modifies the working attribute set.)
- Attributes are altered in the order in which they are specified.
- If the value of an attribute is a Guardian name or a subvolume name, the name is expanded immediately using current default node, volume, and subvolume.
- You cannot alter an attribute unless it is valid for the class of the DEFINE. For example, you cannot alter the FILE attribute if the CLASS of the DEFINE is CATALOG.
- The DEFMODE setting does not affect your ability to alter a DEFINE.

Example—ALTER DEFINE

This example alters the FILE attribute for a set of DEFINES:

```
ALTER DEFINE =CUSTOMER_TABLE, FILE
\SYS1.$TEST.SALES.CUSTOMER;
ALTER DEFINE =ORDERS_TABLE, FILE \SYS1.$TEST.SALES.ORDERS;
ALTER DEFINE =ODETAIL_TABLE, FILE \SYS1.$TEST.SALES.ODETAIL;
ALTER DEFINE =PARTS_TABLE, FILE \SYS1.$TEST.SALES.PARTS;
```

ALTER INDEX Statement

ALTER INDEX is a DDL statement that renames, changes security, or changes file attributes for an entire index; drops or adds an index partition; or changes file attributes for an index partition.

The ALTER INDEX statement also supports several types of move and split operations, including:

- Moving an entire index partition to another disk volume
- Merging an index partition into another existing partition
- Splitting an index partition and moving a part into a newly created partition
- Moving part of an index partition into another existing partition

```
ALTER INDEX name

{
  RENAME new-name
  {
    { security-spec }
    { attribute-spec }
  }
  DROP PARTITION part-name
  {
    {
      { [FROM KEY val [UP TO LAST KEY]]
        TO dest-part [ move-spec ]
        [WITH SHARED ACCESS [ wsa-spec ] ]
      }
      TO dest-part [ move-spec ]
      WITH SHARED ACCESS [ wsa-spec ]
    }
    [PARTONLY] MOVE {
      { ([FROM FIRST KEY] UP TO KEY val
        TO dest-part [ move-spec ],
        [WITH SHARED ACCESS[ wsa-spec ]])
      }
      { ([FROM FIRST KEY] UP TO KEY val
        TO dest-part [ move-spec ],
        FROM KEY val [UP TO LAST KEY]
        TO dest-part [ move-spec ])
      }
    }
    [PARTONLY] {
      {
        { ALLOCATE int | DEALLOCATE }
        MAXEXTENTS int
        RESETBROKEN
        RECOVER INCOMPLETE SQLDDL OPERATION
      }
    }
  }
  ADD PARTITION new-part add-spec
}
```

security-spec is:

```
{ | { CLEARONPURGE | NO CLEARONPURGE } | }
{ | NOPURGEUNTIL date | }
{ | SECURE "rwep" | }
```

attribute-spec is:

```
{ | { ALLOCATE int | DEALLOCATE } | }
{ | { AUDITCOMPRESS | NO AUDITCOMPRESS } | }
{ | { BUFFERED | NO BUFFERED } | }
{ | LOCKLENGTH int | }
{ | MAXEXTENTS int | }
{ | RESETBROKEN | }
{ | { SERIALWRITES | NO SERIALWRITES } | }
{ | TABLECODE int | }
{ | { VERIFIEDWRITES | NO VERIFIEDWRITES } | }
```

move-spec is:

```
{ | CATALOG catalog-name | }
{ | PHYSVOL volume-name | }
{ | EXTENT { size1 | ( size1 [, size2 ] ) } | }
{ | MAXEXTENTS int | }
{ | { FORMAT 1 | FORMAT 2 } | }
{ | DSLACK percent | }
{ | ISLACK percent | }
{ | SLACK percent | }
```

wsa-spec is:

```
{ | NAME operation-name | }
{ | REPORT [ TO collector | ON | OFF ] | }
{ | { COMMIT [ WORK ] [commit-options] } | }
{ | { ROLLBACK [ WORK ] | } | }
```

add-spec is:

```
{ | FIRST KEY { val | ( val [, val ] ... ) } | }
{ | WITH DATA MOVEMENT | }
{ | CATALOG catalog-name | }
{ | PHYSVOL volume-name | }
{ | EXTENT { size1 | ( size1 [, size2 ] ) } | }
{ | MAXEXTENTS int | }
{ | { FORMAT 1 | FORMAT 2 } | }
```


name

is the name of an index or an index partition to alter or move (or an equivalent DEFINE). If *name* is a partition and you use clauses that apply to an entire index, SQL interprets *name* as identifying all partitions of the index.

RENAME *new-name*

changes the file and subvolume portions of the name of an index (including all partitions) to those in the Guardian name (or equivalent DEFINE) *new-name*, updating all catalog references to the index to reflect the change.

name and *new-name* must have the same node and volume name when expanded. If the index is managed by SMF, only the virtual name changes; the physical name on the physical volume is preserved.

security-spec

sets these security-related file attributes for index *name*:

CLEARONPURGE	Controls disk erasure when index is dropped
NOPURGEUNTIL	Sets date after which drop is allowed
SECURE	Sets Guardian security string

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

attribute-spec

sets these file attributes for index *name*:

ALLOCATE	Controls amount of disk space allocated
AUDITCOMPRESS	Controls whether unchanged columns are included in audit records
BUFFERED	Turns buffering on or off
LOCKLENGTH	Sets number of leading bytes in the key to use for generic locks. Default is 0, which specifies the entire key
MAXEXTENTS	Sets maximum extents
RESETBROKEN	Resets BROKEN flag
SERIALWRITES	Specifies serial or parallel writes
TABLECODE	Sets tablecode
VERIFIEDWRITES	Controls verification of writes to disk

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

DROP PARTITION *part-name*

specifies the name (or an equivalent DEFINE) of an empty partition to drop from an index. *part-name* cannot be the primary partition. You cannot drop a partition that

contains data. If you want to drop a partition that contains data, use the PARTONLY option of the PURGEDATA command. For more information on index issues, see [PURGEDATA Command](#) on page P-36.

```
[PARTONLY] MOVE { [FROM KEY val [UP TO LAST KEY]]
                  TO dest-part [ move-spec ]
                  [WITH SHARED ACCESS [ wsa-spec ] ]
                  TO dest-part [ move-spec ]
                  WITH SHARED ACCESS [ wsa-spec ]
                  ( [FROM FIRST KEY] UP TO KEY val
                    TO dest-part [ move-spec ] ,
                    [WITH SHARED ACCESS [ wsa-spec ] ] )
                  ( [FROM FIRST KEY] UP TO KEY val
                    TO dest-part [ move-spec ] ,
                    FROM KEY val [UP TO LAST KEY]
                    TO dest-part [ move-spec ] ) }
```

moves a specified portion of the index *name* to a new or existing partition, *dest-part*, that has the attributes described in *move-spec*. You can specify *dest-part* with a Guardian name, a DEFINE equivalent to a Guardian name, a node name and a volume name, or a volume name only.

For more information about move operations, see the discussion of performing moves under [Considerations—ALTER INDEX](#) on page A-19.

PARTONLY indicates that *name* is a partitioned index, one partition of which is to be moved or split. You must include PARTONLY if *name* is partitioned.

val is a list of comma-separated literals (one for each column in the key) that specifies a point to split the existing partition. Use the keyword NULL to represent a null value in the list.

move-spec

sets the catalog name and these file attributes for the index or partition *dest-part*:

CATALOG	Sets the catalog name
PHYSVOL	Sets a physical volume for the new partition that overrides SMF
EXTENT	Sets extent sizes
MAXEXTENTS	Sets maximum extents
FORMAT	Sets format. Only valid on a node running version 350 or later.
DSLACK	Sets percent of slack in data blocks
ISLACK	Sets percent of slack in index blocks
SLACK	Sets percent of slack in blocks if not specified by DSLACK or ISLACK

The EXTENT, DSLACK, ISLACK, SLACK, and FORMAT options are not supported for a move or a merge of a partition into an existing partition.

name specifies a valid partition of the index. NonStop SQL/MP determines the actual source partition during an execution.

Check that the values you specify result in an index or a partition large enough to hold data being moved from an existing index or partition. Error 45 (File is full) occurs if the new partition does not have enough space to store the rows transferred. The default is the value of the corresponding attribute for the index or partition being moved or split. For more information, see the entry for a specific attribute.

The CATALOG option in *move-spec* specifies a catalog on the same node as *dest-part* to contain the description of the *dest-part*. The default is the current default catalog. If you do not specify CATALOG for a move or a merge operation, SQL determines the correct catalog.

If SMF is installed on your node, the PHYSVOL option directs SQL to override SMF and place the partition on the physical *volume-name*. For *volume-name*, specify either a physical volume or equivalent DEFINE. Do not include the node name in your volume name.

```

                                [ NAME operation-name ]
                                [ ]
WITH SHARED ACCESS [ REPORT [ TO collector | ON | OFF ] ]
                                [ ]
                                [ { COMMIT [WORK] [commit-options] } ]
                                [ { ROLLBACK [WORK] } ]

```

specifies that the partition being moved be accessible for read and write access by DML statements throughout most of the move operation.

The optional clauses allow you to name the operation, control EMS reporting for the operation, specify a time window for the beginning of the commit phase of the operation (the phase in which DML and utility operations on the file are temporarily restricted), and specify the timeout period for lock requests and the handling of retryable errors during the commit phase of the operation.

You can use WITH SHARED ACCESS only if the partition being moved is audited and resides (both before and after the move) on a node running version 315 or later of NonStop SQL/MP. You cannot use WITH SHARED ACCESS within a user-defined transaction.

For information about operations that use WITH SHARED ACCESS, see [WITH SHARED ACCESS OPTION](#) on page W-4. For more information about optional clauses, see [NAME Option](#) on page N-2, [REPORT Option](#) on page R-3, or [COMMIT Option](#) on page C-52.

```
[PARTONLY] { { ALLOCATE int | DEALLOCATE }
              { MAXEXTENTS int
              { RESETBROKEN
              { RECOVER INCOMPLETE SQLDDL OPERATION }
```

changes the ALLOCATE, MAXEXTENTS, RESETBROKEN attributes, or requests a recovery operation to change the INCOMPLETE SQLDDL OPERATION flag for the partition specified in *name*. The keyword PARTONLY is optional and has no effect.

For more information about ALLOCATE, MAXEXTENTS, or RESETBROKEN, see [ALLOCATE File Attribute](#) on page A-7, [MAXEXTENTS File Attribute](#) on page M-2, or [RESETBROKEN File Attribute](#) on page R-20, respectively. For more information about INCOMPLETE SQLDDL OPERATION, see the consideration on page [A-24](#).

ADD PARTITION *new-part add-spec*

adds a partition named *new-part* to index *name* using the options specified in *add-spec*. When specifying *new-part*, include the volume, subvolume, and file name of the partition.

If SMF is installed on your node, the volume can be a virtual or a direct volume. If you specify only a subvolume and a file name, SQL creates a new index partition in the current default volume. If you specify a virtual volume, SQL creates a new index partition in the virtual volume. In all other cases, SQL creates a new index partition in the physical volume and the new partition is a direct file not managed by SMF.

The ADD PARTITION clause is equivalent to the one-way split form of the MOVE clause.

add-spec

specifies options for a partition added with the ADD PARTITION clause.

The FIRST KEY clause is required and specifies the primary or clustering key value for the first key allowed in the new partition. *val* is a literal compatible with the data type of the key column that specifies the key value. For clustering keys, specify multiple *vals*, in order.

The WITH DATA MOVEMENT clause directs SQL to transfer appropriate rows from *name* to *dest-part*. If you do not specify WITH DATA MOVEMENT, ADD PARTITION creates an empty partition and returns an error if records exist within the FIRST KEY declaration of the new partition.

add-spec also includes options that allow you to specify a catalog for the new partition, a physical volume if SMF is installed, the partition's format (1 or 2) and to set the EXTENT and MAXEXTENTS file attributes for the partition. These options are the same as options described under *move-spec* earlier in this entry.

Considerations—ALTER INDEX

- To alter an index, you must be a generalized owner of the index and the underlying table. In addition, you must have authority to read and write to the affected catalogs.

ALTER INDEX executes only if the specified index or partition is accessible. Unless you are altering file attributes for a partition, all partitions of the index must be accessible.

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 an ALTER INDEX statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For more information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

Additional authorization and access requirements that exist for some ALTER INDEX operations are described in the coming subsections.

- You cannot use ALTER INDEX in a user-defined transaction if the index is not audited, if the index is audited and the operation requires data movement, or if the operation is a simple move or a two-way split.
- You cannot rename an index within a user-defined transaction.
- Altering file attributes for indexes
 - To alter security attributes for an index, both the index and its underlying table must be accessible.

You should normally avoid altering the SECURE attribute of an index independent of the SECURE attribute for the underlying table. Restricting read and write access to an index by users who have access to the underlying table can cause some queries on the table (those that use the index) to fail for security reasons while other queries on the same table by the same users succeed. (Note that SQL automatically changes the SECURE and OWNER attributes for an index when you change them for the underlying table, but not the reverse.)

- Changing the SECURE attribute of an index can affect processes using the index when the change occurs. Such processes can continue using the index when they have it open, but might not be able to reopen the index after closing it.

Changing the OWNER for a table automatically changes the OWNER of indexes and protection views defined on the table.

- ALTER INDEX changes file attributes for an index by changing information in the file label for the index and in the catalog tables of the catalog that describes the index.

- MOVE can perform a simple move, a merge (into an existing partition), a one-way or two-way split (to new partitions) or a one-way move (to an existing partition). Note that an index or an index partition cannot be open, even for read access, during an ALTER INDEX MOVE operation, or else the operation fails.

- A simple move moves the partition to another volume:

```
MOVE TO dest-part [WITH SHARED ACCESS]
```

name specifies the partition being moved. You can specify a simple move with or without the WITH SHARED ACCESS option.

- A merge operation moves the partition into another existing partition, deleting the original partition:

```
MOVE TO dest-part WITH SHARED ACCESS
```

name specifies the actual partition being moved. The WITH SHARED ACCESS option is required.

- A one-way split moves the first or last part of a partition to a new partition, leaving the remaining part in the existing partition:

```
MOVE FROM KEY val TO dest-part [WITH SHARED ACCESS]
```

```
MOVE UP TO KEY val TO dest-part WITH SHARED ACCESS
```

In a one-way split, *name* specifies a partition of the index. The partition that is split is the one whose data range would include the key *val*, even if the index partition does not actually contain a row with that key.

For a one-way split operation, the subvolume name and the simple file name for the new partition (whether specified explicitly or by default) must be identical to the subvolume name and the simple file name for every other partition of the same object. ALTER INDEX uses those names if you specify only a node name and a volume name or specify only a volume name (which causes the node to default to the local node). The combination of node name and volume name must be unique for each partition of the same object.

The first part can be moved only if you include the WITH SHARED ACCESS option.

A one-way split without the WITH SHARED ACCESS option requires additional space on the disk that contains the partition being split while the split is in progress. The amount of additional space required can be as much as the size (EOF) of the original partition. A one-way split also requires space within the file itself.

If you are splitting a partition because the disk is full (error 43), it might be permissible to use a two-way split. If you choose to perform a one-way split, use the WITH SHARED ACCESS option. Run a DSAP report to show the largest number of pages of free space. After you successfully perform the one-way split operation, run FUP RELOAD to reclaim unused space in the file. For more information on FUP RELOAD, see the *File Utility Program (FUP)*

Reference Manual. You can also run DCOM on the disk to reclaim unused space. For more information on DCOM, see the *Guardian Disk and Tape Utilities Reference Manual*.

If you are splitting a partition because the file is full (error 45), increase MAXEXTENTS for that file, if possible. Otherwise, perform a two-way split.

If you are splitting a partition because you received SQL error -1637 (“An unexpected error occurred while purging data from filename”), verify that the error was received against the source partition. If the error was on the destination partition, specify smaller extent sizes based on the largest free segment on that disc, or obtain a DSAP report and specify another volume based on the largest free space available on another disk.

- A two-way split moves the first part of a partition to one new partition and the last part of a partition to another new partition, deleting the original partition:

```
MOVE UP TO KEY val TO dest-part
      FROM KEY val TO dest-part
```

In a two-way split, *name* specifies the partition for the operation.

Both occurrences of *val* must be identical, but each occurrence of *dest-part* must specify a different partition. That is, the subvolume name and simple file name for the new partition (whether specified explicitly or by default) must be identical to the subvolume name and simple file name for every other partition of the same object. ALTER INDEX uses those names if you specify only a node name and a volume name or specify only a volume name (which causes the node to default to the local node). The combination of node name and volume name, specified as *dest-part*, must be unique for each partition of the same object.

The two-way split does not support the WITH SHARED ACCESS option.

- A one-way move operation moves the first or the last part of a partition to its logically adjacent partition, leaving the other part in the existing partition. This operation essentially moves the boundary of the partition; for example:

```
MOVE UP TO KEY val TO dest-part WITH SHARED ACCESS
MOVE FROM KEY val TO dest-part WITH SHARED ACCESS
```

A one-way move is similar to a one-way split, but moves data to an existing partition instead of a new partition.

In a one-way move, *name* specifies a valid partition of the index. NonStop SQL/MP determines the actual source partition during execution. The WITH SHARED ACCESS option is required.

After a successful one-way move operation, run FUP RELOAD to reclaim unused disk space. For more information about FUP RELOAD, see the *File Utility Program (FUP) Reference Manual*.

- Reconfiguring partitions of indexes

- All partitions of the index must be accessible when you add a new partition to an index. ALTER INDEX returns an error if you attempt to add a partition while another process has a partition locked or while another process is attempting to execute a DDL operation on the same partition. For more information about the errors returned, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

If you specify WITH SHARED ACCESS, SQL allows concurrent INSERT, UPDATE, DELETE and read-only utility operations on a partition being moved throughout most of the move operation. For more information, see [WITH SHARED ACCESS OPTION](#) on page W-4. You can use the COMMIT option to control when the commit phase occurs and to specify the timeout period for lock requests and the handling of retryable errors (such as errors in lock requests) during the commit phase of the ALTER INDEX operation.

Without WITH SHARED ACCESS, a partition being moved or split is not accessible until the ALTER INDEX operation finishes.

If the partition is being accessed for a SELECT or read operation concurrent with a move or a split operation, the move or the split operation cannot complete until it can obtain an exclusive lock on all partitions. It either waits until the partition becomes available or times out. If the move or split operation obtains an exclusive lock then other transactions against the partition time out.

Other partitions of the table are accessible for INSERT, UPDATE, and DELETE operations, such that processes can make read and write requests for those partitions. For information about specifying on-demand opens, see the OPEN ACCESSED PARTITIONS clause under [CONTROL TABLE Directive](#) on page C-77. Without WITH SHARED ACCESS, you might want to stop activity on a table when you intend to move or split one of the partitions of an index to the table.

If SQL statements refer to the source partition and the partition is moved, you might need to change your program or DEFINES to reference the new location.

- Moving, merging, or splitting an index partition invalidates a program that uses the index, unless the program was compiled with CHECK INOPERABLE PLANS and the table associated with the index has the SIMILARITY CHECK option enabled.

A simple move or a split of a partition invalidates previous TMF online dumps of the affected partition. If you want TMF file-recovery protection, you must make online dumps of the newly moved or split partitions. (If the operation specifies the WITH SHARED ACCESS option, you can begin making new online dumps without waiting for the operation to complete. For more information, see [WITH SHARED ACCESS OPTION](#) on page W-4.)

- Merging a partition or moving all or part of a partition into another existing partition does not change statistics. To update statistics, use the UPDATE STATISTICS command.

- ALTER INDEX operations that use WITH SHARED ACCESS generally take longer to complete than those that do not. However, because WITH SHARED ACCESS operations allow concurrent read and write access to the source partition, they cause far less application downtime than equivalent operations without WITH SHARED ACCESS.

The duration of a WITH SHARED ACCESS operation increases with the number and length of transactions on the node that contains the source partition, particularly with the number and length of transactions that involve the source partition and the amount of activity on the audit trail used for the source partition.

- An operation that uses WITH SHARED ACCESS cannot complete successfully unless the TMF audit trail generated during the operation is accessible for reading later in the operation. If a required audit trail has been overwritten, a WITH SHARED ACCESS operation cancels changes made to the database and terminates.

When performed on a source object that has a valid TMF online dump, an operation that uses WITH SHARED ACCESS generates audit information for the target object.

Lengthy operations that use WITH SHARED ACCESS might require an operator to mount tapes of TMF audit dumps. (Requests to mount TMF audit dump tapes for WITH SHARED ACCESS operations are not distinguishable from other requests to mount TMF audit dump tapes. Such requests are generally sent to an operator's console. SQL does not return information about such requests to the terminal or process that started the operation.)

- When a split command with the WITH SHARED ACCESS option finishes successfully, check SQL FILEINFO for the source partition to see if the F flag is present. For a merge operation with the WITH SHARED ACCESS option, check the target partition; for a one-way move operation with the WITH SHARED ACCESS option, check the source and target partitions. If the F flag is present, the partition contains data blocks allocated to obsolete (moved) records. Use the FUP RELOAD command to reclaim the disk space. For more information, see the *File Utility Program (FUP) Reference Manual*.

If the request fails, the original index normally remains intact and accessible. However, if ALTER INDEX fails because of a processor or system failure, a newly added, moved, or split partition of the index might continue to exist—along with the original index—although it is inaccessible. After the system becomes available, use CLEANUP to drop the new partition (or ask a user with local super ID authority to do so), and reissue the ALTER INDEX statement. ALTER INDEX returns an error if there is a problem with the index.

When you add a partition to an index, the PARTNS catalog table and associated IXPART01 index might become full. To correct the situation, distribute object and partition definitions across multiple catalogs. For more information about partition limits, see [Limits](#) on page L-6.

If ALTER INDEX fails during a merge or one-way move operation with the WITH SHARED ACCESS option, use the SQL FILEINFO utility to see if the D or F flag is present for the target partition:

- The D flag, INCOMPLETE SQLDDL OPERATION, indicates that you need to request an ALTER INDEX *name* PARTONLY RECOVER INCOMPLETE SQLDDL OPERATION, followed by a FUP RELOAD command for the target partition.
- The F flag, UNRECLAIMED FREE SPACE, indicates that you need to request a FUP RELOAD operation to reclaim space from the source partition.

If the slack space in the source table is less than the value chosen for the target table, a MOVE operation can fail with a file full error. To prevent this error, check the actual slack amount in the source file (using FILEINFO STATISTICS) and specify EXTENTS and MAXEXTENTS values for the target table sufficient to hold the data.

- If any partition of an index specified in a move or split operation (even a partition other than the one being moved or split) resides on a node running version 1 of the SQL/MP software, error 1125 (Incompatible remote system) occurs.

You cannot use the WITH SHARED ACCESS option with a split, merge, or move request unless each source object and each target object resides on a node running a version of SQL/MP software (315 or later) that supports the specific type of split, merge, or move operation.

- Dropping partitions of indexes
 - All partitions of an index must be accessible when you drop any partition of the index, but partitions other than the partition being dropped can be accessed by other processes while the ALTER INDEX executes.
 - Dropping an index partition invalidates a program that uses the index, unless the program was compiled with CHECK INOPERABLE PLANS and the table associated with the index has the SIMILARITY CHECK option enabled.
 - If ALTER INDEX fails while attempting to drop a partition, the original index remains intact and accessible.
 - ALTER INDEX cannot drop a primary partition or a partition that is not empty. If you attempt to drop a partition that contains data, SQL returns error 1411 (Operation cannot be performed against a nonempty partition).
- Considerations: Format 2-enabled indexes
 - Indexes inherit their format from the underlying table. An index can be considered to be Format 2-enabled if the underlying table is Format 2-enabled.
 - Existing partitions cannot have their formats altered.
 - When you create a table with a FORMAT2ENABLED partition array, the table will have an object version of at least version 350. Therefore, all partitions of the table and its index partitions must be cataloged in version 350 or later catalogs for the ALTER INDEX catalog to succeed.
- Considerations: ALTER INDEX to add partitions
 - The default partition format for indexes is based on the partition array value of the underlying table. For STANDARD and EXTENDED, the default partition format is 1. For FORMAT2ENABLED, the default partition format is 2. For relative and entry-sequenced tables, the partition format is always 1 because such tables cannot have a FORMAT2ENABLED partition array.
 - The format and file parameters, such as the extent size of the added index partition, must be consistent with the index format for the ALTER INDEX command to succeed. For example, if you explicitly specify a partition as FORMAT 2 when the partition array is STANDARD or EXTENDED you will receive an SQL error.
 - You can use the WITH DATA MOVEMENT clause to move data from an existing partition to the added partition. Data can thus be migrated from Format 1 to Format 2 partitions and from Format 2 to Format 1 partitions.
- Considerations: ALTER INDEX to move partitions
 - The default partition format for indexes is based on the partition array value of the underlying table. For STANDARD and EXTENDED, the default partition format is 1. For FORMAT2ENABLED, the default partition format is 2. For

relative and entry-sequenced tables, the partition format is always 1 because such tables cannot have a FORMAT2ENABLED partition array.

- The format and file parameters, such as extent size of the new index partition, must be consistent with the index format for the ALTER INDEX command to succeed. For example, if you explicitly specify a partition as FORMAT 2 when the partition array is STANDARD or EXTENDED you will receive an error.
- The FORMAT clause can be specified for a simple move, one-way split, or two-way split operation.
- SQL ignores the FORMAT clause if you specify it for a merge or a one-way move (move partition boundary) operation because these types of moves do not create a new partition. If you specify the FORMAT clause on these types of operations you will receive a warning and the operation continues.

Examples—ALTER INDEX

- This example changes the value of MAXEXTENTS and deallocates the disk space for the index XORDCUS:

```
ALTER INDEX \SYS1.$VOL1.SALES.XORDCUS
  MAXEXTENTS 300 DEALLOCATE;
```

- This example changes the maximum number of extents for index XEMPNAME to 200. It also sets the SERIALWRITES attribute to specify serial mirror writes for operations on the index:

```
ALTER INDEX \SYS.$VOL1.PERSNL.XEMPNAME
  MAXEXTENTS 200 SERIALWRITES;
```

- This example renames the index XEMP:

```
ALTER INDEX XEMP RENAME XEMPID;
```

- This example moves an index partition, specifying WITH SHARED ACCESS to keep the partition accessible to other processes during most of the move:

```
ALTER INDEX $DISK1.USERS.XDATA PARTONLY MOVE TO $DISK2
  WITH SHARED ACCESS NAME MOVE2D2 COMMIT BY REQUEST;
```

```
...
CONTINUE MOVE2D2 ONCOMMITERROR COMMIT BY REQUEST;
```

The ALTER INDEX statement specifies COMMIT BY REQUEST so that the user can control entry to the commit phase of the operation, which locks out other processes. The CONTINUE statement starts the commit phase, directing SQL to return control to the user if a retryable error occurs during the phase.

- This example moves the latter portion of an index partition into a new partition (a one-way split):

```
ALTER INDEX =XPART_LOC PARTONLY MOVE
  FROM KEY "I00" TO =XPART_EUROPE
  EXTENT (8,8) SLACK 20;
```

- This example creates two new partitions of an index and moves data to them from an existing partition, which it deletes (a two-way split):

```
ALTER INDEX $DISK1.SALES.XORDERS PARTONLY MOVE
( FROM FIRST KEY UP TO KEY 50 TO $DISK2 CATALOG =CAT2,
  FROM KEY 50 UP TO LAST KEY TO $DISK3 CATALOG =CAT3 );
```

- This example moves the latter portion of an index partition into a new partition (a one-way split) in which the new partition is Format 2:

```
ALTER INDEX =XPART_LOC PARTONLY MOVE
FROM KEY "I00" TO =XPART_EUROPE
EXTENT (8,8) SLACK 20 FORMAT 2;
```

ALTER PROGRAM Statement

ALTER PROGRAM is a DDL statement that renames an SQL program in a Guardian file or alters security-related file attributes for an SQL program in a Guardian file. (You cannot use ALTER PROGRAM on an SQL program in an OSS file.)

```
ALTER PROGRAM program { security-spec }
                       { RENAME new-name }

security-spec is:

{ [NO]CLEARONPURGE
  SECURE "rwep"
  OWNER group-num,user-num [ NO PROGID ]
  { PROGID | NO PROGID } }
```

program

is the name of a file (or an equivalent DEFINE) that contains an SQL-compiled SQL program in a Guardian file.

security-spec

sets these security-related file attributes:

CLEARONPURGE	Controls disk erasure when file is dropped
SECURE	Sets Guardian security string
OWNER	Specifies owner
PROGID	Determines PAID of process from file

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

RENAME *new-name*

renames the program and changes all references to the old name in the affected catalog to the new name; *new-name* is a Guardian name or DEFINE. The fully

expanded new name must be unique among objects in the network. Both *program* and *new-name* must have the same volume and node name.

If the program is managed by SMF, *new-name* must be either a virtual or direct name. Only the virtual name changes; the physical name on the physical volume is preserved.

Considerations—ALTER PROGRAM

- To alter security attributes for a program or rename a program, you must be a generalized owner of the program file. You must also have authority to read and write the program file.

To rename a program, you must also have authority to read and write to the catalogs that describe the program and any associated objects. Renaming a program does not affect the validity of the program.

If the program is protected by the Safeguard security subsystem, requirements depend on the Safeguard protection settings. For example, if access is restricted to the super ID, you must be the super ID or error 199 (Disk file is Safeguard protected) occurs. If you are the super ID, ALTER PROGRAM executes successfully, but the new security attributes take effect only if Safeguard protection is removed from the program.

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 execute an ALTER PROGRAM statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For more information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

- The security attribute information for an SQL program in a Guardian file is stored in the file label and in the PROGRAMS table of the catalog when the program is created. ALTER PROGRAM changes the information in the catalog and in the associated file label on disk.
- These dependencies apply when you alter the program security attributes:

OWNER and PROGID	Specifying OWNER turns off the PROGID attribute.
OWNER and SECURE	A change in the ownership of a program affects the interpretation of the security string. The security string is interpreted at run time against the new owner and, if applicable, a new group. If another process is using a program when the owner or security string is changed, the process might not be able to access the program after the program stops executing.
SECURE	A security string must ensure that users who have write access also have read access.

Example—ALTER PROGRAM

This example statements give the program file ASERV to a new owner (12,201) and set the PROGID file attribute. The first statement sets the owner ID, which automatically turns off the PROGID attribute. The second statement turns on the PROGID attribute.

```
ALTER PROGRAM ASERV OWNER 12,201;  
ALTER PROGRAM ASERV PROGID;
```

ALTER TABLE Statement

ALTER TABLE is a DDL statement that:

- Renames, changes security, changes file attributes, or enables or disables similarity checks for a table
- Adds a column to a table or changes the HEADING attribute for an existing column
- Drops, moves, splits, reuses, or adds a table partition or changes file attributes of a table partition

The ALTER TABLE statement supports several types of move and split operations, including:

- Moving an entire partition to another disk volume
- Merging an entire partition into another existing partition
- Splitting a partition and moving part into a newly created partition
- Moving part of a partition into another existing partition.

ALTER TABLE *name*

```

{
  RENAME new-name
  {
    { security-spec }
    { attribute-spec }
  }
  SIMILARITY CHECK { ENABLE | DISABLE }
  ADD COLUMN col-name data-type [ DEFAULT def
    [ NOT NULL ] ] [ HEADING string | NO HEADING ]
  COLUMN col-name { HEADING string | NO HEADING }
  PARTITION ARRAY
    { STANDARD | EXTENDED | FORMAT2ENABLED }
  DROP PARTITION part-name
    {
      { [FROM KEY val [UP TO LAST KEY]]
        TO dest-part [ move-spec ]
        [WITH SHARED ACCESS [ wsa-spec ] ]
      }
      TO dest-part [ move-spec ]
      WITH SHARED ACCESS [ wsa-spec ]
    }
  [PARTONLY] MOVE
    {
      { ([FROM FIRST KEY] UP TO KEY val
        TO dest-part [ move-spec ]
        [WITH SHARED ACCESS [ wsa-spec ] ])
      }
      { ([FROM FIRST KEY] UP TO KEY val
        TO dest-part [ move-spec ],
        FROM KEY val [UP TO LAST KEY]
        TO dest-part [ move-spec ] )
      }
    }
  [PARTONLY] {
    { ALLOCATE int | DEALLOCATE }
    MAXEXTENTS int
    RESETBROKEN
    RECOVER INCOMPLETE SQLDDL OPERATION
  }
  ADD PARTITION
    new-part add-spec
  REUSE PARTITION reused-part reuse-spec
}

```

security-spec is:

```

{
  { CLEARONPURGE | NOCLEARONPURGE }
  NOPURGEUNTIL date
  OWNER group-num, user-num
  SECURE "rwep"
}

```


attribute-spec is:

```
{ | { ALLOCATE int | DEALLOCATE } | }
{ | { AUDIT | NO AUDIT } | }
{ | { AUDITCOMPRESS | NO AUDITCOMPRESS } | }
{ | { BUFFERED | NO BUFFERED } | }
LOCKLENGTH int
MAXEXTENTS int
RESETBROKEN
{ | { SERIALWRITES | NO SERIALWRITES } | }
TABLECODE int
{ | { VERIFIEDWRITES | NO VERIFIEDWRITES } | }
```

move-spec is:

```
{ | CATALOG catalog-name | }
{ | PHYSVOL volume-name | }
EXTENT { size1 | ( size1 [, size2 ] ) }
MAXEXTENTS int
FORMAT 1 | FORMAT 2
DSLACK percent
ISLACK percent
SLACK percent }
```

wsa-spec is:

```
{ | NAME operation-name | }
{ | REPORT [ TO collector | ON | OFF ] | }
{ | { COMMIT [ WORK ] [commit-options] } | }
{ | { ROLLBACK [ WORK ] | } | }
```

add-spec is:

```
{ | FIRST KEY { val | ( val [, val ] ... ) } | }
{ | WITH DATA MOVEMENT | }
{ | CATALOG catalog-name | }
{ | PHYSVOL volume-name | }
EXTENT { size1 | ( size1 [, size2 ] ) }
MAXEXTENTS int
FORMAT 1 | FORMAT 2 g }
```

reuse-spec is:

```
{ | FIRST KEY { val | ( val [, val ] ... ) } | }
```

name

is the name of the table or table partition to alter or move (or an equivalent DEFINE). If *name* is a partition and you use clauses that apply to a entire table, SQL interprets *name* as identifying all partitions of the table.

name cannot specify a catalog table other than CATALOGS, PROGRAMS, TRANSIDS, or USAGES. (You can use *security-spec* to change security attributes for these tables, but you cannot use any other ALTER TABLE clause with any catalog table.)

RENAME *new-name*

changes the file and subvolume portions of the name of a table (including all its partitions) to those in the Guardian name (or equivalent DEFINE) *new-name*, updating all catalog references to reflect the change.

name and *new-name* must have the same node and volume name when expanded.

security-spec

sets these security-related file attributes for table *name*:

CLEARONPURGE	Controls disk erasure when table is dropped
NOPURGEUNTIL	Sets date after which drop is allowed
OWNER	Specifies owner
SECURE	Sets Guardian security string

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

attribute-spec

sets these file attributes for table *name*:

ALLOCATE	Controls amount of disk space allocated
AUDIT	Controls TMF auditing. Default is AUDIT
AUDITCOMPRESS	Controls whether unchanged columns are included in audit records
BUFFERED	Turns buffering on or off
LOCKLENGTH	Sets byte count in key for generic locks
MAXEXTENTS	Sets maximum extents
RESETBROKEN	Resets BROKEN flag
SERIALWRITES	Specifies serial or parallel writes
TABLECODE	Sets tablecode
VERIFIEDWRITES	Controls verification of writes to disk

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

SIMILARITY CHECK { ENABLE | DISABLE }

authorizes or prohibits similarity checks on table *name*.

Authorizing similarity checks (SIMILARITY CHECK ENABLE) on a table whose version is older than 310 increases the version of the table (and the version of objects that depend on the table) to 310. Such a table cannot be registered in an older version catalog or accessed by older versions of the SQL/MP software.

Prohibiting similarity checks (SIMILARITY CHECK DISABLE) on a table that has version 310 decreases the version of the table (and the version of objects that depend on the table).

```
ADD COLUMN col-name data-type [ DEFAULT def
```

```
[ NOT NULL ] ] [ HEADING string | NO HEADING ]
```

adds a column named *col-name* to table *name*, including all its partitions, if any.

The *data-type* and the DEFAULT clause specify the data type and default value for the new column. For more information about these clauses, see [Data Types](#) on page D-1 and [DEFAULT Clause](#) on page D-26.

NOT NULL specifies that the new column cannot contain null values. If you specify NOT NULL, you must specify the DEFAULT clause also.

The HEADING clause specifies a heading for the new column. For more information, see [HEADING Clause](#) on page H-1.

You cannot add a column to a table with entry-sequenced file organization.

```
COLUMN col-name { HEADING string | NO HEADING }
```

specifies a new default heading for the existing column *col-name* in table *name*. For more information, see [HEADING Clause](#) on page H-1.

```
PARTITION ARRAY { STANDARD | EXTENDED | FORMAT2ENABLED }
```

specifies the type of partition array used for the specified table and all associated indexes:

EXTENDED	specifies the extended partition array available on versions 320 and later of NonStop SQL/MP
STANDARD	specifies the type of array used by default by NonStop SQL/MP
FORMAT2ENABLED	specifies the size of the partition array, and designates whether the table is Format 1 enabled or Format 2 enabled. STANDARD and EXTENDED designate a Format 1-enabled table. FORMAT2ENABLED designates a Format 2-enabled table. Only key sequenced files can be Format 2 enabled.

An extended partition array supports a larger number of table and index partitions. It also allows more indexes to be created against the base table. A

Format 2-enabled partition array allows a partition to grow to one terabyte or the size of a single disk volume, whichever is smaller.

PARTITION ARRAY applies to partitions created later for a table, even if the table is not currently partitioned. Altering the base table causes all associated indexes to be altered automatically to the value specified for the base table.

You can use the PARTITION ARRAY clause in SQLCI or in dynamic SQL statements. To check its value, use the FILEINFO DETAIL command.

For more information, see [Modifying the partition array](#) on page A-47.

DROP PARTITION *part-name*

specifies the name of an empty partition to drop from a table. *part-name* cannot be the primary partition. you cannot drop a partition that contains data. You cannot drop a partition that contains data unless you use the PARTONLY option of PURGEDATA. For more information on index issues, see [PURGEDATA Command](#) on page P-36.

```
[PARTONLY] MOVE {
  { [FROM KEY val [UP TO LAST KEY]]
    TO dest-part [ move-spec ]
    [WITH SHARED ACCESS [wsa-spec]] } }
  TO dest-part [ move-spec ]
  WITH SHARED ACCESS [wsa-spec]
  ( [FROM FIRST KEY] UP TO KEY val
    TO dest-part [move-spec]
    [WITH SHARED ACCESS [wsa-spec]]) }
  ( [FROM FIRST KEY] UP TO KEY val
    TO dest-part [ move-spec ],
    FROM KEY val [UP TO LAST KEY]
    TO dest-part [ move-spec ] ) }
```

moves a specified portion of the table *name* to a new or existing partition, *dest-part*, that has the attributes described in *move-spec*.

You can specify *dest-part* with a Guardian name, a DEFINE equivalent to a Guardian name, a node name and volume name, or a volume name only.

For more information about types of move and split operations, see the discussion of performing moves under [Considerations—ALTER INDEX](#) on page A-19.

PARTONLY indicates that *name* is a partitioned table, one partition of which is to be moved or split. You must include PARTONLY if *name* is partitioned.

val is a list of comma-separated literals (one for each column in the key) that specifies a point at which to split the existing partition.

move-spec

sets the catalog name and these file attributes for the table or partition *dest-part*:

CATALOG	Sets the catalog name
PHYSVOL	Sets a physical volume for the new partition that overrides SMF
EXTENT	Sets extent sizes
MAXEXTENTS	Sets maximum extents
FORMAT 1 FORMAT 2	Sets partition format.
DSLACK	Sets percent of slack in data blocks
ISLACK	Sets percent of slack in index blocks
SLACK	Sets percent of slack in blocks if not specified by DSLACK or ISLACK

The DSLACK, ISLACK, SLACK, EXTENT and FORMAT clauses are not supported for a move or merge of a partition into an existing partition.

Make sure that the values you specify result in a table or partition large enough to hold data being moved from an existing table or partition. Error 45 (File is full) occurs if the new partition does not have enough space to store the rows transferred. The default is the value of the corresponding attribute for the table or partition being moved or split. For more information, see the entry for a specific attribute.

The CATALOG option in *move-spec* specifies a catalog on the same node as *dest-part* to contain the description of the *dest-part*. The default is the current default catalog. If you do not specify CATALOG for a one-way move or a merge operation, SQL determines the correct catalog.

If SMF is installed, the PHYSVOL option directs SQL to override SMF and move the partition to the physical volume-*name*. For *volume-name*, specify either a physical volume or equivalent DEFINE. Do not include the node name in your volume name.

```

[ NAME operation-name ]
[
WITH SHARED ACCESS [ REPORT [ TO collector | ON | OFF ] ]
[
[ {COMMIT [WORK] [commit-options]} ]
[ {ROLLBACK [WORK]} ] ] ]

```

specifies that the partition being moved be accessible for read and write access by DML statements throughout most of the move or split operation.

The option clauses allow you to name the operation, control EMS reporting for the operation, specify a time window for the beginning of the commit phase of the operation (the phase in which DML and utilities operations on the file are temporarily restricted), and specify the timeout period for lock requests and the handling of retryable errors during the commit phase of the operation.

You can use WITH SHARED ACCESS only if the partition being moved or split is audited and resides (both before and after the operation) on a node running version 315 or later of NonStop SQL/MP. You cannot use WITH SHARED ACCESS within a user-defined transaction.

For information about operations that use WITH SHARED ACCESS, see [WITH SHARED ACCESS OPTION](#) on page W-4. For information about the optional clauses, see [NAME Option](#) on page N-2, [REPORT Option](#) on page R-3, or [COMMIT Option](#) on page C-52.

```
[PARTONLY] { { ALLOCATE int | DEALLOCATE }
              { MAXEXTENTS int
              { RESETBROKEN
              { RECOVER INCOMPLETE SQLDDL OPERATION }
            }
```

changes the ALLOCATE, MAXEXTENTS, or RESETBROKEN attribute, or requests a recovery operation to change the INCOMPLETE SQLDDL OPERATION flag for the partition specified in *name*. The keyword PARTONLY is optional and has no effect.

For more information about ALLOCATE, MAXEXTENTS, or RESETBROKEN, see the [ALLOCATE File Attribute](#) on page A-7, [MAXEXTENTS File Attribute](#) on page M-2, or [RESETBROKEN File Attribute](#) on page R-20 entries. For more information about INCOMPLETE PARTITION CHANGE, see the [Considerations—ALTER TABLE](#) on page A-37.

ADD PARTITION *new-part add-spec*

adds a partition named *new-part* to table *name* using the options specified in *add-spec*. When specifying *new-part*, include the volume, subvolume, and file name of the partition.

If SMF is installed, the volume can be a virtual or direct volume name. If you specify only a subvolume and file name, SQL creates a new index partition in the current default volume. If you specify a virtual volume, SQL creates a new index partition in the virtual volume. In all other cases, SQL creates a new index partition in the physical volume and the new partition is a direct file not managed by SMF.

You must use ADD PARTITION to add a partition to a table with relative or entry-sequenced file organization. For tables with key-sequenced organization, you can use the MOVE clause to add a partition; ADD PARTITION is equivalent to the one-way split form of the MOVE clause.

add-spec

specifies options for a partition added with the ADD PARTITION clause.

The FIRST KEY clause specifies the primary or clustering key value for the first key allowed in a new partition of a table with key-sequenced file organization. It is required for key-sequenced files. *val* is a literal compatible with the data type of the key column that specifies the key value. For clustering keys, specify multiple *vals*, in order.

The FIRST KEY clause does not apply to tables with relative or entry-sequenced file organization; ADD PARTITION adds an empty partition to the end of a relative or entry-sequenced file.

The WITH DATA MOVEMENT clause directs SQL to transfer appropriate rows from *name* to *dest-part*. If you do not specify WITH DATA MOVEMENT, ADD PARTITION creates an empty partition and returns an error if records exist within the FIRST KEY declaration of the new partition.

add-spec also includes options that allow you to specify a catalog for the new partition, a physical volume if DSM/Storage Manager is installed, and to set the EXTENT, MAXEXTENTS and FORMAT file attributes for the partition. These options are the same as options described under *move-spec* earlier in this entry.

REUSE PARTITION *reused-part reuse-spec*

reuses a partition, specified in *reused-part*, using the options specified in *reuse-spec*. When specifying *reused-part*, include the volume, subvolume, and file name of the partition.

reuse-spec

specifies FIRST KEY for a partition reused with the ADD PARTITION clause.

The FIRST KEY clause specifies the primary or clustering key values for the first key allowed in a reused partition of a table with key-sequenced file organization. It is required for key-sequenced files. *val* is a list of comma-separated literals (one for each column in key) that specifies the beginning key value for the partition to be reused.

Considerations—ALTER TABLE

- To alter a table or partition, you must be a generalized owner of the table or partition. In addition, you must have authority to read and write to the affected catalogs. For a partitioned table, affected catalogs include all catalogs that describe a partition of the table or that will describe a new partition of the table as a result of the ALTER TABLE operation.

ALTER TABLE executes only if the specified table or partition is available. Unless you are altering file attributes for a partition, all partitions of the table must be available.

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 execute an ALTER TABLE statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For more information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

Additional authorization and access requirements that exist for some ALTER TABLE operations are described in the coming subsections.

- Renaming tables
 - To rename a table, you must have authority to read and write to the catalogs that describe the views, indexes, programs, and collations associated with the table, in addition to the catalog that describes the table.
 - Renaming a table does not invalidate any programs dependent on the table. Existing OPENS of the renamed table are redirected to the new name, and the corresponding programs continue to run until the new OPENS are required. The non-running programs dependent on the table might experience subsequent errors unless:
 - A replacement copy of the original table name is created, or
 - DEFINES are re-directed to the renamed table. SQL re-compilation might be necessary or advisable in some situations.
 - You cannot rename a table within a user-defined transaction.
- Altering file attributes for tables
 - To alter security attributes for a table, the table and any indexes and protection views defined on the table must be accessible.
 - Changing either the SECURE or OWNER attribute of a table can affect processes using the table when the change occurs. ALTER TABLE closes the table to make the change. Other SQL processes using the table attempt to reopen it after the change, but the new attribute values can prevent such processes from doing so.
 - Changing the OWNER for a table automatically changes the OWNER of indexes and protection views defined on the table.
 - Changing the SECURE attribute for a table automatically changes the SECURE attribute of indexes defined on the table. In addition, changing the SECURE attribute for a table automatically changes the SECURE attribute of protection views defined on the table if such a change is necessary to meet these requirements:
 - Purge authority for the protection view must include the users authorized to purge the underlying table.

- The owner of a protection view must have authority to read and write to the view and the underlying table unless the security string for the view specifies the super ID (-) for the authority the owner lacks.

SQL issues a warning if it changes the security string of protection views as the result of an ALTER TABLE.

- Changing the SECURE attribute read authority for a table can effectively change the read authority for a shorthand view defined on the table, because authority to read a shorthand view depends on the authority to read the underlying tables. (The read option in the SECURE value for a shorthand view has no effect.)
- Altering the AUDIT attribute of a table automatically changes the AUDIT attribute value for dependent views and indexes to the same value. In addition, altering the AUDIT attribute of a table automatically changes the BUFFERED attribute for the table (but not for dependent views and indexes):

AUDIT Attribute	BUFFERED Attribute
AUDIT	BUFFERED
NO AUDIT	NO BUFFERED

You can override the automatic change to the BUFFERED attribute by explicitly setting BUFFERED in the ALTER TABLE.

If you change an indexed, unaudited table to an audited table, you can create performance problems unless you also modify the BUFFERED attribute for the indexes on the table in separate ALTER INDEX statements. The AUDIT and BUFFERED attributes on a table and its indexes should be set to the same value.

An ALTER TABLE that changes the AUDIT attribute to NO AUDIT cannot execute in a user-defined TMF transaction.

- ALTER TABLE changes file attributes for a table by changing information in the file label for the table and in the catalog tables of the catalog that describes the table.
- Adding columns
 - ALTER TABLE ADD COLUMN automatically requests an exclusive table lock on the table. If data in the table is already locked, ALTER TABLE waits until the request is granted or a timeout occurs.
 - A new column appears as the last column of the table. In existing rows of the table, the new column takes on its default value unless it has a date-time data type with the default set to CURRENT or SYSTEM.

If you add a column with a date-time data type to a table that contains existing rows and you specify DEFAULT CURRENT or DEFAULT SYSTEM, SQL uses January 1, 1 A.D. 12:00:00.000000 as the default date and time for the existing rows.

For example, an existing row receives the value 0001-01-01:12:00:00.000000 in the new column if the data type is DATETIME YEAR TO FRACTION, receives the value 0001-01-01 in the new column if the data type is DATE, receives the value 12:00:00 in the new column if the data type is TIME, and so forth.

Any row added after the ADD COLUMN operation finishes that does not contain a value for the column receives a default value based on the current timestamp at the time the row is added.

- The sum of the lengths of all columns for a table cannot exceed the maximum row length for the table (the block size minus the header size). For information about additional restrictions on the number of columns allowed, see [Limits](#) on page L-6.
- You cannot add a column to a table with relative file organization unless the row length of the table is large enough to accommodate the added column. You cannot add a column to a table with entry-sequenced file organization under any circumstances.
- The new column is not actually added to a row until the row is updated. If you select a row that does not yet have the new column, SQL returns the default value for the column.
- MOVE can perform a simple move operation for a table of any file organization, and can perform a one-way or two-way split (to new partitions), merge (into an existing partition), or one-way move (to an existing partition) for a table with a key-sequenced file organization.
 - A simple move moves a partition to another volume:


```
MOVE TO dest-part [WITH SHARED ACCESS]
```

name specifies the name of the partition to be moved. You can specify a simple move with or without the WITH SHARED ACCESS option.
 - A merge operation moves the partition into another existing partition, deleting the original partition:


```
MOVE TO dest-part WITH SHARED ACCESS
```

In a merge request, *name* specifies the actual partition being moved. The WITH SHARED ACCESS option is required.
 - A one-way split moves the first or last part of a partition to a new partition, leaving the remaining part in the existing partition:


```
MOVE FROM KEY val TO dest-part [WITH SHARED ACCESS]
MOVE UP TO KEY val TO dest-part WITH SHARED ACCESS
```

In a one-way split, *name* specifies the table for the operation. The partition that is split is the one whose data range includes the key *val*, even if the specified partition does not actually contain a row with that key.

For a one-way split, the subvolume name and simple file name for the new partition (whether specified explicitly or by default) must be identical to the subvolume name and simple file name for every other partition of the same object; ALTER TABLE uses those names if you specify only a node name and volume name or specify only a volume name (which causes the node to default to the local node). The combination of node name and volume name must be unique for each partition of the same object.

The first part can be moved only if you include the WITH SHARED ACCESS option.

A one-way split without the WITH SHARED ACCESS option requires additional space on the disk that contains the partition being split while the split is in progress. The amount of additional space required can be as much as the size (EOF) of the original partition. A one-way split also requires space within the file itself.

If you are splitting a partition because the disk is full (error 43), it might be permissible to use a two-way split. If you choose to perform a one-way split, use the WITH SHARED ACCESS option. Run a DSAP report to show the largest number of pages of free space. After you successfully perform the one-way split operation, run FUP RELOAD to reclaim unused space in the file. For more information on FUP RELOAD, see the *File Utility Program (FUP) Reference Manual*. You can also run DCOM on the disk to reclaim unused space. For more information on DCOM, see the *Guardian Disk and Tape Utilities Reference Manual*.

If you are splitting a partition because the file is full (error 45), increase MAXEXTENTS for that file, if possible. Otherwise, perform a two-way split.

If you are splitting a partition because you received SQL error -1637 (“An unexpected error occurred while purging data from filename”), verify that the error was received against the source partition. If the error was on the destination partition, specify smaller extent sizes based on the largest free segment on that disc, or obtain a DSAP report and specify another volume based on the largest free space available on another disk.

- A two-way split moves the first part of a partition to one new partition and the last part of a partition to another new partition, deleting the original partition:

```
MOVE UP TO KEY val TO dest-part
      FROM KEY val TO dest-part
```

In a two-way split, *name* specifies the partition for the operation.

Both occurrences of *val* must be identical, but each occurrence of *dest-part* must specify a different partition. That is, the combination of node name and volume name for a new partition (whether specified explicitly as *dest-part* or by default) must be unique for each partition of the same object. The subvolume name and simple file name for a new partition (whether specified explicitly or by default) must be identical to the subvolume name and

simple file name for every other partition of the same object. ALTER TABLE uses those names if you specify only a node name and volume name or specify only a volume name (which causes the node to default to the local node).

The two-way split does not support the WITH SHARED ACCESS option.

- A one-way move operation moves the first or last part of a partition to its logically adjacent (existing) partition, leaving the other part in the existing partition:

```
MOVE UP TO KEY val TO dest-part WITH SHARED ACCESS
MOVE FROM KEY val TO dest-part WITH SHARED ACCESS
```

A one-way move is similar to a one-way split, but moves data to an existing partition instead of a new partition.

In a one-way move, *name* specifies a valid partition of the table. NonStop SQL/MP determines the actual source partition during execution.

The WITH SHARED ACCESS option is required.

After a successful one-way move operation, run FUP RELOAD to reclaim unused disk space. For more information about FUP RELOAD, see the *File Utility Program (FUP) Reference Manual*.

- All partitions of the table must be accessible when you add a new partition to a table. ALTER TABLE returns an error if you attempt to add a partition while another process has a partition locked or while another process is attempting to execute a DDL operation on the same partition. For more information about the errors returned, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

When you move the primary partition of a table, you must have read and write authority for its associated catalogs, indexes, views, and programs.

If you specify WITH SHARED ACCESS, SQL allows concurrent INSERT, UPDATE, DELETE, and read-only utility operations on a partition being moved throughout most of the move operation. For more information, see [WITH SHARED ACCESS OPTION](#) on page W-4. You can use the COMMIT option to control when the commit phase occurs and to specify the time-out period for lock requests and the handling of retryable errors (such as errors in lock requests) during the commit phase of the ALTER TABLE operation.

Without WITH SHARED ACCESS, a partition being moved or split is not accessible for INSERT, UPDATE, or DELETE operations until ALTER TABLE finishes, but is available for SELECT operations during most of the DDL operation. If the partition is being accessed for a SELECT or read operation, the move or split operation cannot complete until it can obtain an exclusive lock on all partitions, and so it either waits until the partition becomes available or times out.

Other partitions of the table are available for INSERT, UPDATE, or DELETE operations, so processes can make read and write requests for those partitions. For information about specifying on-demand opens, see the OPEN ACCESSED PARTITIONS clause and the SKIP UNAVAILABLE PARTITION clause under [CONTROL TABLE Directive](#) on page C-77.

Without WITH SHARED ACCESS, you might want to stop activity on a table when you intend to move or split one of the partitions to the table.

To update label information for partitions, ALTER TABLE requires exclusive label locks on all partitions of a table during the final phase of a move or split operation. Transaction activity on the table can cause ALTER TABLE to time out when it attempts to acquire the locks. Alternatively, ALTER TABLE can successfully acquire the locks while other transactions are active and cause those transactions to time out.

If SQL statements refer to the source partition and the partition is moved with a simple move operation, you might need to change your program or DEFINES to reference the new location.

- If you add, move, or split a partition to a table that has a protection view defined on it, SQL automatically creates a corresponding partition for the protection view on the same subvolume as the table partition and writes the view description in the same catalog as the table partition description.

Moving, merging, or splitting a table partition invalidates a program that uses the table or a dependent view, unless the program was compiled with CHECK INOPERABLE PLANS and the table or view has the SIMILARITY CHECK option enabled.

A simple move or a split of a partition invalidates previous TMF online dumps of the affected partition. If you want TMF file recovery protection, you must make online dumps of the newly moved or split partitions. (If the operation specifies the WITH SHARED ACCESS option, you can begin making new online dumps without waiting for the operation to complete. For more information, see the [WITH SHARED ACCESS OPTION](#) on page W-4.)

- Merging a partition or moving all or part of a partition into another existing partition does not change related items such as statistics, comments, and help text. To update statistics, use the UPDATE STATISTICS command. Check your comments and help text to make sure they still apply.
- The first key value of a new partition cannot duplicate the first key value of another partition of the table.

You cannot create a partition on a nonaudited volume, even if the table that includes the partition is a nonaudited table.

New partitions must comply with the limits on the number and size of partitions. For more information, see [Limits](#) on page L-6.

You can partition tables of any file organization, but you cannot partition a key-sequenced table that has a system-defined primary key (as opposed to a user-defined primary key) unless it also has a clustering key.

- ALTER TABLE operations that use WITH SHARED ACCESS generally take longer to complete than those that do not. However, because WITH SHARED ACCESS operations allow concurrent read and write access to the source partition, they cause far less application downtime than equivalent operations without WITH SHARED ACCESS.

The duration of a WITH SHARED ACCESS operation increases with the number and length of transactions on the node that contains the source partition, particularly with the number and length of transactions that involve the source partition and the amount of activity on the audit trail used for the source partition.

- An operation that uses WITH SHARED ACCESS cannot complete successfully unless the TMF audit trail generated during the operation is available for reading later in the operation. If a required audit trail has been overwritten, a WITH SHARED ACCESS operation cancels changes made to the database and terminates.

When performed on a source object that has a valid TMF online dump, an operation that uses WITH SHARED ACCESS generates audit information for the target object.

Lengthy operations that use WITH SHARED ACCESS might require an operator to mount tapes of TMF audit dumps. (Requests to mount TMF audit dump tapes for WITH SHARED ACCESS operations are not distinguishable from other requests to mount TMF audit dump tapes. Such requests are generally sent to an operator's console. SQL does not return information about such requests to the terminal or process that started the operation.)

- When a split operation with the WITH SHARED ACCESS option finishes successfully, check SQL FILEINFO for the source partition to see if the F flag is present. For a merge operation with the WITH SHARED ACCESS OPTION, check the target partition; for a one-way move operation with the WITH SHARED ACCESS option, check the source and target partitions. If the F flag is present, the file contains data blocks allocated to obsolete (moved) records. Use the FUP RELOAD command to reclaim the disk space. For more information, see the *File Utility Program (FUP) Reference Manual*.

If ALTER TABLE fails, the original table normally remains intact and accessible. However, if ALTER TABLE fails because of a processor failure or system crash, a newly added, moved, or split partition of the table might continue to exist—along with the original table—although it is inaccessible. After the system becomes available, use CLEANUP to drop the new partition (or ask the local super ID to do so), reissue the ALTER TABLE statement. ALTER TABLE returns an error if there is a problem with the table.

When you add a partition to a table, the PARTNS catalog table and associated IXPART01 index might become full. To correct the situation, distribute object and partition definitions across multiple catalogs. For more information about partition limits and the PARTNS table, see [Limits](#) on page L-6.

If the SLACK space in the source file is less than the value chosen for the target file, a MOVE operation can fail with a file full error. To prevent this situation, check the actual slack amount in the source file (using the FILEINFO STATISTICS command) and specify EXTENTS and MAXEXTENTS values for the target file that are sufficient to hold the data.

If ALTER TABLE fails during a merge or move operation with the WITH SHARED ACCESS option, use the SQL FILEINFO utility to see if the D or F flag is present for the target partition:

- The D flag, INCOMPLETE SQLDDL OPERATION, indicates that you need to issue an ALTER TABLE *name* PARTONLY RECOVER INCOMPLETE SQLDDL OPERATION request, followed by FUP RELOAD.
- The F flag, UNRECLAIMED FREE SPACE, indicates that you need to request a FUP RELOAD operation to reclaim space on disk.
- You cannot use the WITH SHARED ACCESS option with a split, merge, or move request unless each source object and each target object resides on a node running a version of the SQL/MP software (315 or later) that supports the specific type of split, merge, or move operation.

If any partition of a table specified in a move or split operation (even a partition of the table other than the one being moved or split) resides on a node running version 1 of the SQL/MP software, error 1125 (Incompatible remote system) occurs.

- Considerations: Format 2-enabled tables

- The partition array can be modified between any of the partition array values.
- Modifying the partition array of a table implicitly converts all indexes on the table to the same partition array value. There is no command to explicitly alter the partition array of an index because an index inherits the value from the table. Modifying a table to allow it to have a FORMAT2ENABLED partition array results in the partition arrays of all the indexes of that table being converted to FORMAT2ENABLED.
- A table can only be converted to a Format 1 from a Format 2 table if all partitions of the table and all partitions of all indexes of the table are Format 1. If you attempt to convert a Format 2 to Format 1 when any Format 2 partitions exist you will receive an error.
- Modifying the partition array does not change the format of any of the existing partitions. It enables new partitions to be created in formats compatible to that partition array value.

- Similar to other DDL operations, all partitions of the table must be available for the operation to complete.
- Only key-sequenced tables can be Format 2-enabled tables. If you attempt to specify `FORMAT2ENABLED` for a relative or entry-sequenced table, you receive an error.
- If you attempt to convert a Format 1-enabled table to Format 2-enabled where the existing number of partitions is greater than that supported by Format 2, you will receive an error. Because modifying the partition array of a table also implicitly modifies the partition arrays of the indexes, the indexes on the table are subject to the same limits. An error is returned if any indexes violate the limits.
- Modifying the partition array of a table to be `FORMAT2ENABLED` causes the object version of the table and all its indexes to become at least version 350. Therefore, all partitions of the table and its indexes must be cataloged in version 350 or later catalogs to modify the partition array to `FORMAT2ENABLED`. Modifying the partition array of a table to be `STANDARD` or `EXTENDED` causes the object version of the table to be recalculated based upon the other features associated with the table.
- When converting a table from Format 1 enabled to Format 2 enabled, the row length of the table or an index on the table might not fit within a Format 2 block because the amount of overhead space for the block header is larger for Format 2 files. The `ALTER TABLE` statement fails when this condition occurs.
- Considerations: `ADD PARTITION` to specify partition format
 - The default partition format is based on the partition array value for key-sequenced tables. For `STANDARD` and `EXTENDED`, the default partition format is 1. For `FORMAT2ENABLED`, the default partition format is 2. For relative and entry-sequenced tables, the partition format is always 1.
 - The format and file parameters such as extent size of the added partition must be consistent with the existing table for the `ALTER TABLE` command to succeed. For example, if you explicitly specify the new partition as `FORMAT 2` when the partition array is `STANDARD` or `EXTENDED` you will receive an error.
 - The `WITH DATA MOVEMENT` clause can be used to move data from an existing partition to the added partition. Data can be migrated from Format 1 to Format 2 partitions and from Format 2 to Format 1 partitions.
 - Only key-sequenced tables can be Format 2 enabled. Relative and entry-sequenced tables can only be Format 1 enabled. If you attempt to add a Format 2 partition to a relative or entry-sequenced table you will receive an error. Specifying `FORMAT 1` for a relative or entry-sequenced partition is allowed.
 - The `FORMAT` clause is ignored if specified for a merge or one-way move (move partition boundary) operation because these types of moves do not

create a new partition. If you specify the `FORMAT` clause on these types of operations you will receive a warning and the operation continues.

- Dropping partitions of tables

- All partitions of a table must be accessible when you drop any partition of the table, but partitions other than the partition being dropped can be accessed by other processes while the `ALTER TABLE` executes.

Dropping a partition requires SQL to update any catalog that describes a shorthand view in terms of the dropped partition, and to update the catalogs for the table itself. If any such catalog is unavailable, the `ALTER TABLE` fails.

- Dropping a table partition invalidates a program that uses the table or that uses a view that depends on the table, unless the program was compiled with `CHECK INOPERABLE PLANS` and the table or view has the `SIMILARITY CHECK` option enabled.

Invalid programs that reference the table with `DEFINE`s or with references to partitions other than the dropped partition can be recompiled (automatically or explicitly). Programs that reference the table with a physically coded name for the dropped partition must be modified before they can be recompiled.

- If `ALTER TABLE` fails while attempting to drop a partition, the original table remains intact and accessible.
- `ALTER TABLE` cannot drop a primary partition, a partition that is not empty, or a partition in a table with relative or entry-sequenced file organization that is not the last partition.

- Modifying the partition array

- Tables and indexes using extended arrays require a catalog of version 320 or later. SQL DML and DDL statements on tables and indexes with extended arrays can only be performed from nodes running version 320 or later. Otherwise, SQL returns an error.

When you modify the partition array of a table, all programs that reference the table are invalidated. Recompile the programs with NonStop SQL/MP version 320 or later.

When you alter a table from `EXTENDED` to `STANDARD`, the data structures might not fit within the `STANDARD` format. When this occurs, SQL returns an error.

- Reusing partitions

- Use `REUSE PARTITION` instead of dropping and adding partitions to manage and reuse disk space. `DROP/ADD` can take hours to perform if the partition being reused is from a table with many partitions.
- `REUSE PARTITION` purges all the records in the existing partition. It modifies the key range of the partition in the label with a new key range to

accommodate database growth. Because each partition must have access to the location and key range of other partitions, REUSE PARTITION updates the label of all the partitions with the new key range.

- After REUSE PARTITION is performed, the reused partition is empty. If an error is encountered during REUSE, the data in the reused partition might not be recoverable. It is recommended that you back up data before performing REUSE PARTITION.
- During the operation of reusing a partition, the partition becomes inaccessible for read and write by DML statements until the operation is done.
- Only a partition with key-sequenced file organization can be reused. A partition with relative or entry-sequenced file organization cannot be specified in REUSE.
- REUSE PARTITION can only be performed against the secondary partition of a partitioned table. It cannot be performed on a primary partition or any partitioned indexes.
- REUSE PARTITION cannot be executed within a user transaction. This prevents a user from performing a rollback of the transaction during the REUSE. Instead, it is executed within the transaction started by the catalog manager.
- A partitioned table with or without view dependent objects can be reused. A table on which indexes are defined cannot be reused.
- If records exist within the FIRST KEY specified, SQL returns an error. The partition is not used and the partitions remain intact. The key ranges of each partition cannot overlap.

Examples—ALTER TABLE

- This example alters the security and owner for a table:

```
ALTER TABLE \SYS1.$VOL2.PERSNL.EMPLOYEE
  SECURE "nunu" OWNER 12,101;
```

- This example alters the date that a table can be purged:

```
ALTER TABLE SALES.ORDERS NOPURGEUNTIL JAN 01 2004;
```

- This example turns on TMF auditing (perhaps after temporarily setting the NO AUDIT attribute to perform a LOAD operation) for a table:

```
ALTER TABLE SALES.ORDERS AUDIT;
```

- This example adds a column to a table:

```
ALTER TABLE CUSTOMER ADD COLUMN LAST_ORDER_DATE
  NUMERIC(6) DEFAULT 860000 HEADING "Last Ordered";
```

- This example alters the maximum number of extents for a specific partition:

```
ALTER TABLE \SYS2.$VOL2.INVENT.PARTLOC
PARTONLY MAXEXTENTS 300;
```

- This example drops a table partition (after deleting all rows):

```
DELETE FROM =PARTLOC
WHERE LOC_CODE >= "H00" AND LOC_CODE < "K00";
ALTER TABLE =PARTLOC DROP PARTITION =PART_SOUTH;
```

- This example moves a table partition to another disk, specifying WITH SHARED ACCESS to keep the table available to other process during the move:

```
ALTER TABLE $DISK1.USERS.DATA PARTONLY MOVE TO $DISK2
WITH SHARED ACCESS NAME MOVE2D2 COMMIT BY REQUEST;
...
CONTINUE MOVE2D2 ONCOMMITERROR COMMIT BY REQUEST;
```

The ALTER TABLE statement specifies COMMIT BY REQUEST so that the user can control entry to the commit phase of the operation, which locks out other processes. The CONTINUE statement starts the commit phase, directing SQL to return control to the user if a retryable error occurs during the phase.

- This example moves the latter portion of a table partition into a new partition (a one-way split):

```
ALTER TABLE =PART_LOC PARTONLY MOVE
FROM KEY (RH00S, 0) TO =PART_EUROPE EXTENT (8,8);
```

- This example creates two new partitions of a table and moves data to them from an existing partition, which it deletes (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);
```

- This example reuses a partition named \$DISK5.SALES.ORDERS, in which the new primary first key is 5000 after the partition is reused:

```
ALTER TABLE $DISK1.SALES.ORDERS REUSE PARTITION
$DISK5.SALES.ORDERS FIRST KEY "5000"
```

- This example moves the latter portion of a table partition into a new partition (a one-way split) in which the new partition is Format 2:

```
ALTER TABLE =PART_LOC PARTONLY MOVE
FROM KEY (RH00S, 0) TO =PART_EUROPE EXTENT (8,8) FORMAT 2;
```

ALTER VIEW Statement

ALTER VIEW is a DDL statement that changes the name, owner, or security attributes of a view, changes the heading for a column of the view, or enables or disables similarity checks for the view.

```
ALTER VIEW view-name

{
  RENAME new-name
}

{
  {
    OWNER group-num,user-num
    SECURE "rwep"
  }
}

COLUMN col-name { HEADING string | NO HEADING }

SIMILARITY CHECK { ENABLE | DISABLE }
```

view-name

is the name of a view to alter (or an equivalent DEFINE).

RENAME *new-name*

changes the file and subvolume portions of the name of a view to those in the Guardian name (or equivalent DEFINE) *new-name*, updating all catalog references to reflect the change. If SMF is installed, *new-name* must be either a virtual or direct name.

The fully expanded *new-name* must be unique among object names in the network and *new-name* must have the same volume name as *view-name*. If the view is managed by SMF, only the virtual name changes; the physical name on the physical volume is preserved.

OWNER *group-num,user-num*

specifies the Guardian user ID for the new owner of the view.

SECURE "*rwep*"

specifies the new Guardian security string for the view. For more information, see [Security](#) on page S-11.

COLUMN *col-name* { HEADING *string* | NO HEADING }

specifies a default heading or no heading for the column *col-name*. For more information, see [HEADING Clause](#) on page H-1.

Changing a column heading in a view does not change the corresponding heading in any view previously created on that view. A view inherits headings from underlying tables and views when you create it, but the headings exist independently.

`SIMILARITY CHECK { ENABLE | DISABLE }`

authorizes or prohibits similarity checks on a protection view. (You cannot specify this clause for a shorthand view.)

Authorizing similarity checks (`SIMILARITY CHECK ENABLE`) on a view whose version is older than 310 increases the version of the view (and the version of objects that depend on the view) to 310. Such a view cannot be registered in an older version catalog or accessed by older versions of NonStop SQL/MP.

Prohibiting similarity checks (`SIMILARITY CHECK DISABLE`) on a view that has version 310 decreases the version of the view (and the version of objects that depend on the view).

Considerations—ALTER VIEW

- To rename a view or alter security attributes for a view, you must be a generalized owner of the view. You must also have authority to read the view.

You must have authority to read and write to the catalogs that describe the view and any associated objects. If you rename a view, all dependent programs are invalidated.

If a view to be renamed is a protection view, the underlying table and indexes must be accessible and you must have read and write authority.

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 execute an `ALTER VIEW` statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For more information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

- Changing an attribute of a view can change the version of the view (and the version of objects that depend on the view).
- These dependencies apply when you alter view security attributes:

OWNER and
SECURE

A change in the ownership of a view affects the interpretation of the security string. The security string is interpreted at run time against the new owner and, if applicable, a new group.

If another process is using a view when the owner or security string is changed, the process might not be able to access the view after the view is closed.

SECURE

A security string must ensure that users who have write access also have read access.

Example—ALTER VIEW

These four examples rename, resecure, change the owner of, and change a column heading for a view,

```
ALTER VIEW NAME1 RENAME NAME2;
ALTER VIEW NAME2 SECURE "nnno";
ALTER VIEW NAME2 OWNER 12,72;
ALTER VIEW NAME2 COLUMN DESCR HEADING "Product/Descriptions";
```

APPEND Command

APPEND is an SQLCI utility that appends data from an SQL table or Guardian file (such as a Guardian process, device, unstructured disk file, or Enscribe file) to an entry-sequenced or key-sequenced SQL table. Neither type of SQL table can have indexes. APPEND adds data to the end of a table or partition. Unlike SQLCI LOAD, it does not overwrite data in the target table.

-
- △ **Caution.** When appending data to an entire table, APPEND requires that you turn off auditing for the table. Doing so invalidates TMF online dumps of the table. To ensure TMF volume recovery protection for the table, turn AUDIT back on when the APPEND operation is complete, and make new TMF online dumps of all table partitions.

When you use the PARTONLY option to append data to a single partition, you do not need to turn off auditing, as the command does it for you. You still need to make an online dump of the single partition.

```
APPEND in-file, out-file, RECOVERYFILE recovery-file
      [ [,] [append-option] ] ... ;
```

in-file

is the name (or equivalent DEFINE) of the table or file from which to append data. *in-file* can be a Guardian process, device (such as a terminal or tape), unstructured disk file, Enscribe file, or SQL table. For an SQL table and a non-SQL object, setting the PARTONLYIN option directs SQL to load from the partition specified as *in-file*.

out-file

is the name (or equivalent DEFINE) of an existing entry-sequenced or key-sequenced SQL table to which to append data. Neither type of file can have indexes.

recovery-file

is the name (or equivalent DEFINE) of a disk file APPEND creates to hold data for recovery in case of interruption or failure in restoring *out-file* to its pre failure state. RECOVERYFILE is a required parameter and must be the third parameter in the command string

append-option

is one or more options that configure the APPEND operation. The *append-option* list is identical to *load-option* in the LOAD command, except you cannot APPEND data to an Enscribe file. Therefore, Enscribe-specific LOAD options are not valid for an APPEND operation. For a full description of each *append-option*, see [LOAD Command](#) on page L-18.

Considerations—APPEND

- APPEND requires authority to read *in-file* and write to *out-file*. You must also have authority to read the catalogs in which *in-file* (if it is a table) and *out-file* are described.

recovery-file inherits the default file creation security of the user who executes APPEND. Because APPEND copies some data from *out-file* to *recovery-file*, if *out-file* contains sensitive information and the APPEND operation fails, the security of this information can be compromised. To protect sensitive information during an APPEND operation, set the user's default file creation *recovery-file* security appropriately. You can change default file creation security temporarily in the TACL VOLUME command or permanently in the TACL DEFAULT command. For information about these commands, see the *TACL Reference Manual*.

- For an entry-sequenced target table, APPEND adds new data to the end of the table. If the table is not partitioned, it is unavailable to applications during the APPEND operation. If the table is partitioned, the partition into which the first new row was added and all subsequent partitions are unavailable during the APPEND operation. The whole table is unavailable for a short time at the start of APPEND, so this is not really an online operation.

For a key-sequenced target table, the key value of every new row to add must be logically greater than the key value of the last original row of the table. If you append data to only one partition (APPEND PARTONLY), the key value of every new row to add must be logically greater than the key value of the last original row of the partition and logically less than the starting key value of the next partition. To interleave new rows with original rows, use COPY.

If it is not partitioned, a key-sequenced target table is unavailable to applications during the APPEND operation. If the table is partitioned and you do not specify PARTONLY, the partition into which the first new row was added and all subsequent partitions are unavailable. The whole table is unavailable for a short time at the start of APPEND, so this is not really an online operation. If you do specify PARTONLY, only the partition to which APPEND adds data is unavailable during the APPEND operation.

To append data to several partly-full partitions of a table, execute an APPEND PARTONLY command for each partition, giving each its own input file.

- A recovery file contains information needed to restore the target table to its original state if the APPEND operation fails. Before appending any data to a table or table partition, APPEND checks for the presence of a recovery file. If the file does not exist or exists but contains incomplete recovery data, the APPEND command executes normally, as described below. If the file exists and contains recovery data or is not a recovery file, APPEND fails. In the normal case, APPEND executes:
 - Creates a new *recovery-file*
 - Gathers information about the current state of *out-file*
 - Writes this information to *recovery-file*
 - Appends data from *in-file* to *out-file*
 - Purges *recovery-file* in these situations:
 - The APPEND operation completes successfully
 - The APPEND operation terminates with an error but APPEND successfully restores the target table to its original state
- The APPEND operation can terminate without completing its task under two general conditions:
 - An error causes the APPEND operation to terminate gracefully.
 If the APPEND operation terminates with an error, it attempts to use the recovery information in *recovery-file* to restore the target table to its original state (that is, the state the table was in before the APPEND operation was started). If APPEND succeeds in restoring the target table to its initial state, the APPEND operation purges *recovery-file*.

 In this case, the APPEND operation did not successfully complete, and no *recovery-file* exists. Thus, the absence of *recovery-file* after an APPEND operation does not always mean that the operation was successful. To determine if the APPEND operation succeeded, check the SQLCI listing to see if error messages occurred.
 - A processor or process failure interrupts the APPEND operation so that it cannot terminate gracefully.

 If the APPEND operation is interrupted and cannot restore the target table to its original state, it does not purge *recovery-file*. In this case, the target table or partition remains unavailable to applications until some recovery is performed.

 To recover from an interrupted APPEND operation and complete the operation, use the APPENDRESTART command. You must specify exactly the same parameters, including the same *recovery-file*, as you did in the initial APPEND command.

 To recover from an interrupted APPEND operation and simply restore the target table to its original state, use the APPENDCANCEL command. After an

APPENDCANCEL operation executes, the target table contains exactly the same data it had before the initial APPEND operation was started. (Any new data added by the interrupted APPEND operation is removed from the target table.)

For more information about these commands, see [APPENDRESTART Command](#) on page A-58 and [APPENDCANCEL Command](#) on page A-56.

- If you repeat an APPEND command that terminated successfully on an entry-sequenced table, the target table will contain duplicate data. Do not repeat an APPEND command unless you want to create duplicate data in the target table.
- APPEND resembles COPY in that both transfer data from an existing source to an existing target without overwriting or erasing target data.

Major differences between APPEND and COPY:

- APPEND is faster than COPY.
- APPEND cannot run within a user-defined TMF transaction. COPY can run within a user-defined TMF transaction.
- APPEND writes only to entry-sequenced and key-sequenced SQL tables. COPY writes to all types of SQL tables in addition to Enscribe files.
- APPEND does not write to SQL tables with indexes. COPY writes to SQL tables with indexes and automatically updates the indexes.
- APPEND adds data only to the end of a table or partition. COPY can interleave new rows with existing rows.
- APPEND resembles LOAD in that both transfer data from an existing source to an existing target. Neither APPEND nor LOAD can run within a user-defined TMF transaction, and both require that you turn off auditing when transferring data to an entire table. Like LOAD, APPEND starts an external sort process to sort the data unless you specify the SORTED option. All considerations listed for LOAD that do not require an Enscribe *out-file* are true for APPEND.

Major differences between APPEND and LOAD:

- APPEND is typically used to add data to a file that already contains data. LOAD is typically used to enter initial data into an empty file.
- APPEND does not erase or overwrite existing records. LOAD erases or overwrites existing records.
- APPEND writes only to entry-sequenced and key-sequenced SQL tables. LOAD writes to all types of SQL tables and Enscribe files.
- APPEND does not write to SQL tables with indexes. LOAD allows the target to have indexes and re-creates indexes for SQL tables.
- APPEND runs normally and reports no errors when *in-file* is empty. LOAD terminates and reports an error when the input file is empty.

- Field Conversions

For any APPEND operation, the data type of each source field must be compatible with the data type of its corresponding target field. The details of data type compatibility and Enscribe-to-SQL and SQL-to-Enscribe field conversions are identical for APPEND and LOAD; see [Data type compatibility and field conversions](#) on page L-36 for details.

APPENDCANCEL Command

APPENDCANCEL is an SQLCI utility that recovers from an interrupted APPEND operation by restoring the target table to its original state (before the APPEND operation began). APPENDCANCEL deletes any data appended to the target table by the interrupted APPEND operation. You should use the APPENDCANCEL command only after an APPEND operation is interrupted by a processor or process failure, or other unexpected termination.

```
APPENDCANCEL out-file, RECOVERYFILE recovery-file
                [ [,] PARTONLY ] ;
```

out-file

is the name (or equivalent DEFINE) of an existing entry-sequenced or key-sequenced SQL table without indexes whose APPEND is to be canceled. *out-file* must be identical to the *out-file* specified in the interrupted APPEND command from which this APPENDCANCEL command is recovering.

recovery-file

is the name (or equivalent DEFINE) of a disk file APPENDCANCEL uses to restore *out-file* to its original state. RECOVERYFILE is a required parameter. *recovery-file* must be identical to the *recovery-file* specified in the interrupted APPEND command from which this APPENDCANCEL command is recovering.

If APPENDCANCEL is successful, then it purges the *recovery-file*.

PARTONLY

You must use the PARTONLY option if PARTONLY was specified in the original APPEND command from which this APPENDCANCEL command is recovering.

Considerations—APPENDCANCEL

- APPENDCANCEL has the same authorization requirements as the APPEND command. For more information about authorization requirements for APPENDCANCEL, see [APPEND Command](#) on page A-52. APPENDCANCEL verifies that the information in *recovery-file* accurately describes the target table (or partition). Using the information in *recovery-file*, APPENDCANCEL restores the target table to the state it was in before the APPEND operation began.
- APPENDCANCEL verifies that the information in *recovery-file* accurately describes the target table (or partition). Using the information in *recovery-file*, APPENDCANCEL restores the target table to the state it was in before the APPEND operation began.

After the APPENDCANCEL operation finishes, the target table or partition is accessible to applications again.

Suppose that an APPEND operation finishes appending data to the target table and is interrupted at the end of the operation, when it is clearing the corrupt flags. In this case, you cannot use the APPENDCANCEL command to recover from the interrupted APPEND operation. (After the APPEND operation has begun clearing corrupt flags, the target table cannot safely be restored to its original state.) In this situation, the APPENDCANCEL command returns an error 9179 and has no effect; use the APPENDRESTART command instead. For more information, see [Considerations—APPENDRESTART](#) on page A-59.

- The APPEND operation creates a recovery file specified by *recovery-file*. During normal operation, the APPEND utility purges *recovery-file* after it successfully finishes adding data to the end of the target table.

However, if a processor or process failure interrupts the APPEND operation, *recovery-file* is not purged and should continue to exist and be available for APPENDCANCEL. One exception is if APPEND is interrupted at the end of the operation as described under the previous item. Another is if the interruption occurs early in the APPEND operation, before APPEND finishes creating *recovery-file*. In that case, you cannot use APPENDCANCEL to recover from the failure.

If APPENDCANCEL returns error 9182, indicating it cannot complete its operation because the original APPEND did not complete the *recovery-file*, you should purge the partial *recovery-file*. Do NOT purge a *recovery-file* unless an error 9182 is generated to indicate that the recover-file was not completed by the original APPEND.

APPENDRESTART Command

APPENDRESTART is an SQLCI utility that recovers from an interrupted APPEND operation and completes the APPEND operation. The APPENDRESTART utility restores the target table to its original state (before the APPEND operation began). APPENDRESTART proceeds with the APPEND operation, adding data to the end of the target table. You should use the APPENDRESTART command only after an APPEND operation is interrupted by a processor or process failure, or other ungraceful termination.

```
APPENDRESTART in-file, out-file,
              RECOVERYFILE recovery-file
              [ [,] [append-option] ] ... ;
```

in-file

is the name (or equivalent DEFINE) of the table or file from which to append data. *in-file* can be a Guardian process, device (such as a terminal or tape), unstructured disk file, Enscribe file, or SQL table. *in-file* must be identical to the *in-file* specified in the interrupted APPEND command from which this APPENDRESTART command is recovering. This match is checked only for disk files or tables.

out-file

is the name (or equivalent DEFINE) of an existing entry-sequenced or key-sequenced SQL table to which to append data. Neither type of file can have indexes. *out-file* must be identical to the *out-file* specified in the interrupted APPEND command from which this APPENDRESTART command is recovering.

recovery-file

is the name (or equivalent DEFINE) of a disk file APPENDRESTART uses to restore *out-file* to its original state. RECOVERYFILE is a required parameter. *recovery-file* must be identical to the *recovery-file* specified in the interrupted APPEND command from which this APPENDRESTART command is recovering.

If APPENDRESTART is successful, it purges *recovery-file*.

append-option

is one or more options that configure the APPENDRESTART operation. The options included in *append-option* must be identical to those specified in the interrupted APPEND command from which this APPENDRESTART command is recovering. For more information about *append-option*, see [APPEND Command](#) on page A-52.

Considerations—APPENDRESTART

- APPENDRESTART has the same authorization requirements as the APPEND command. For more information about authorization requirements for APPENDRESTART, see [APPEND Command](#) on page A-52.
- APPENDRESTART verifies that the information in *recovery-file* accurately describes both the source file and the target table (or partition). Using the information in *recovery-file*, APPENDRESTART restores the target table to the state it was in before the APPEND operation began. APPENDRESTART proceeds with the original APPEND operation, appending data from the source file to the end of the restored target table.

After the APPENDRESTART operation finishes, the target table or partition is accessible to applications again.

The APPENDRESTART operation is likely to take as long as the original APPEND operation, even if the interruption occurred when the APPEND operation was, for example, halfway finished. This operation time occurs because APPENDRESTART starts over at the beginning of the APPEND operation, adding all the data to the target table that would have been added by APPEND.

In one case, however, APPENDRESTART is likely to execute in a very short time. Suppose that an APPEND operation finishes writing data to the target table and is interrupted at the end of the operation, when it is clearing corrupt flags. The subsequent APPENDRESTART operation recognizes that all the data has been successfully appended. In that case, APPENDRESTART does not restore the target table to its original state. Instead, it simply finishes clearing corrupt flags and completes the operation.

- The APPEND operation creates a recovery file specified by *recovery-file*. During normal operation, the APPEND utility purges *recovery-file* after it successfully finishes adding data to the end of the target table.

However, if a processor or process failure interrupts the APPEND operation, *recovery-file* is not purged and should continue to exist and be available for APPENDRESTART; the only exception is if the interruption occurs early in the APPEND operation, before APPEND finishes creating *recovery-file*. In that case, you cannot use APPENDRESTART to recover from the failure. Instead, use the APPEND command again. APPEND accepts an existing *recovery-file* if it is incomplete, and writes fresh recovery information to *recovery-file*.

AS Clause

The AS clause is an SQLCI report writer clause that specifies a display format for a print item. You can use the AS clause in the BREAK FOOTING, BREAK TITLE, DETAIL, PAGE FOOTING, PAGE TITLE, REPORT FOOTING, and REPORT TITLE report writer commands.

AS	<pre> { display-descriptor } "scale-sign-descriptor display-descriptor" } " "[" [decoration [, decoration] ...] [modifier [, modifier] ...] "]" display-descriptor" } " "[" [decoration [, decoration] ...] [modifier [, modifier] ...] "]" (scale-sign-descriptor display-descriptor)" } </pre>
----	--

display-descriptor

specifies a format.

scale-sign-descriptor

specifies scale for a fixed point display descriptor or controls printing of a plus sign.

decoration

specifies character strings to conditionally add to a print item.

modifier

alters the effect of descriptors.

These tables in this entry describe descriptors for character items, descriptors for numeric items, scale-sign descriptors, decorations, and modifiers, respectively:

- [Display Descriptors for Character Items](#) on page A-61
- [Display Descriptors for Numeric Items](#) on page A-61
- [Scale-Sign Descriptors](#) on page A-62
- [Modifiers](#) on page A-62
- [Decorations](#) on page A-64

For a description of how to format dates and times in Julian timestamps, see [AUDIT File Attribute](#) on page A-74.

Table A-2. Display Descriptors for Character Items

Form and Usage	Example	Value	Printed
A[w] Character field, <i>w</i> print positions wide. Default for <i>w</i> is width of item. If too small, left justify and blank fill. If too large, truncate.	A A4 A3 “[LJ] A3” “[RJ] A3”	WORD WORD WORD WORD WORD	WORD WORD WOR WOR ORD
Cn[.w] Multiline character field, <i>n</i> print positions wide with <i>w</i> print positions per line. If <i>n</i> is 0, use width of item; if <i>n</i> is more than 255, you must specify <i>w</i> .	C0.8	Customer has a low credit rating. (VARCHAR string)	Customer has a l ow credi t rating .
To break a value at blanks, use the F modifier with a <i>Cn</i> descriptor.	“[F] C40.8”	A reliable vendor - fast delivery	A reliable vendor - fast delivery

w is an unsigned integer in the range 1-255
n is an unsigned integer in the range 1-4071
 || indicates the boundaries of the output field
 LJ and RJ are modifiers (covered later in this entry) that specify left and right justification

PIC 9 data type is numeric and should not use the A[w] descriptor. [Table A-3](#) describes descriptors that you can use with PIC 9.

A print position is one byte, so a double-byte character (which occupies two print positions) requires a descriptor width of at least twice its number of characters. For example, a double-byte column with ten characters requires an A20 display descriptor.

Table A-3. Display Descriptors for Numeric Items (page 1 of 2)

Form and Usage	Example	Value	Printed
Fw.d [.m] Fixed point field, <i>w</i> print positions wide with <i>d</i> significant digits to the right of the decimal and <i>m</i> to the left. Right justify, blank fill, leading zeros if needed.	F8.4 F8.4 F8.4.2 “[FL*]F8.2”	123.4567 0.000123 - 4.56789 123.4567	123.4567 0.0001 -04.5679 **123.46
lw [.m] <i>d</i> , <i>w</i> , and <i>n</i> are unsigned integers in the range 1-255. indicates the boundaries of the output field.			

Table A-3. Display Descriptors for Numeric Items (page 2 of 2)

Form and Usage	Example	Value	Printed
Integer field, w print positions wide that contains m digits. Right justify, blank fill, leading zeros.	I8	100	100
	I8.2	-1	-01
	I8.6	100	000100
M			
Mask string in angle brackets, apostrophes, or quotes is template for display. Field is same width as mask, excluding V's. Mask does not affect scale.	M"99/99/99"	112388	11 / 23 / 88
	M'Z,ZZ9.99'	32.009	32.01
	M<Z,ZZZ>	666	666
	M<9,999>	666	0,666
	M<9,999>	66666	*****
	M<\$Z,ZZ9.99>	920.00	\$ 920.00
	M<\$Z,ZZ9V99>	920.00	\$ 92000
Ordinary characters in the mask appear in the display. These characters have special meanings:			
Z	Selects digits. Prints a blank for each leading or trailing zero. Use with character or numeric data. Must be uppercase Z.		
9	Selects digits. Prints zero if no digit exists. Use for numeric data only.		
V	Aligns the decimal point of the value but does not print a decimal point. Must be uppercase V.		
.	Aligns the decimal point of the value and prints a decimal point (the character specified in the DECIMAL_POINT style option) at that position.		
d, w, and n are unsigned integers in the range 1-255.			
indicates the boundaries of the output field.			

Table A-4. Scale-Sign Descriptors

Form and Usage	Example	Value	Printed
nP			
Scale factor 10**n. n is -128 to 127.	"2P F8.2"	123.00	12300.00
	"-2P F8.2"	123.00	1.23
S or SS (default) Omit plus sign.	"S, F8.2"	123.00 123.00	123.00
	"SS,-2P F8.2"		1.23
SP			
Print plus sign.	"SP, F8.2"	123.00	+123.00
	"SP, F8.2"	-123.00	-123.00
indicates the boundaries of the output field			
An S, SS, or SP scale-sign descriptor must be separated from a display descriptor by a comma.			

Table A-5. Modifiers (page 1 of 2)

Form and Usage	Example	Value	Printed
BZ			
Prints a blank field if the value is zero.	"[BZ] F8.2"	.00	
	"[BZ] F8.2"	123.00	123.00
F			

Table A-5. Modifiers (page 2 of 2)

Form and Usage	Example	Value	Printed
Specifies whether data in C format is split at a blank if possible. FL'c'	"[F] C24.8"	Manager is on leave.	Manager is on leave.
Specifies an ASCII character to use when data has fewer characters than specified by an alphanumeric descriptor, when leading zeros are suppressed, or when mask text is not printed because adjacent digits are not printed. The default is a blank. OC'c'	"[FL'.'] A8" "[RJ,FL'>']A8" "[FL'*)]M<\$Z,ZZ9.99"	THEN HERE 127.39	THEN... >>>HERE \$**127.39
Specifies an ASCII character to use as the overflow character for the current print item. (Does not apply to alphanumeric descriptors.) LJ	"[OC'+']I2" "[OC'>']F5.2"	100 100000.0 0	++ >>>>
Left justify in display format width. Applies only to A descriptors; default for A descriptors. Note that data with leading blanks will not look justified. RJ	"[LJ]A8" "[LJ]A3"	SQL OREGON	SQL ORE
Right justify in display format width. Applies only to A descriptors. Note that data with trailing blanks won't look justified. SS'ss'	"[RJ]A8" "[RJ]A3"	SQL OREGON	SQL GON
Changes a descriptor symbol. First character in pair is old symbol, second is new. Use to change mask descriptor symbols (9,Z,V,.) and F descriptor decimal point (.) symbol.	"[SS'.']F6.2" "[SS'.']F8.2" "[SS'9X'] M<XX/XX/19XX>"	12.45 12345.67 103191	12:45 12345,67 10/31/1991

Decorations

decoration

is *condition location char-string*

condition is one or more of:

- M Add *char-string* if value is negative
- P Add *char-string* if value is positive
- Z Add *char-string* if value is zero
- O Add *char-string* if overflow condition occurs

Only P is allowed with character print items.

Specify multiple conditions without separators.

location

specifies where the character prints:

- A_n* Display at print position *n* of the field. (The leftmost position is 1.)
- F Insert string after formatting value; print string immediately to the right of a left-justified item, immediately to the left of a right-justified item.
- P Insert string before formatting value; print string immediately to the right of a right-justified item, immediately to the left of a left-justified item.

char-string

is a string add to the print item.

Rules for Using Decorations

- Specify decorations immediately before the descriptors they modify. Enclose the decorations and the modified descriptors in quotation marks ("").
- Separate multiple decorations by commas and enclose decorations and modifiers in brackets.
- With the overflow condition (O), use only the form *location A_n*.
- The default overflow character is the current OVERFLOW_CHAR option. Decoration OC overrides the default but does not change it.
- A print position is one byte; each double-byte character requires two print positions.
- Without decorations, a negative value prints with a preceding minus. If you specify a decoration that tests for a positive value, however, the preceding minus is no longer the default for negatives; you must explicitly request the negative sign.

Order for Processing Decorations

SQL processes decorations from left to right:

1. Tests data to determine whether it is positive, negative, or zero.
2. If the P location is specified, adds the character string to the item value.
3. Formats according to the A, I, or F descriptor.
4. Applies decorations for alphanumeric and fixed-point descriptors.
5. Tests for overflow.

Examples—AS

- Some simple uses of the AS clause:

CUSTNAME AS "[RJ] A24"	Displays CUSTNAME right justified in a 24-byte field
CUSTNAME AS A24	Displays CUSTNAME left justified in a 24-byte field
PRICE AS "-2P F6.0"	Displays PRICE in a 6-byte field with a scale of minus 2 and no digits to the right of the decimal
JOBDESC AS "[F] C18.10"	Displays an 18-byte varying-length character column as ten bytes per line, breaking lines at blanks if possible
- This AS clause displays a dollar value, enclosing the value in angle brackets if the value is negative:

```
QTYCOST AS "[MF'<',MP'>',ZPP' ' ] (-3P F10.2)"
```

Sample positive display: | 46983.00 |

Sample negative display: | <240.00> |

- These are examples of decorations for negative, positive, zero, and overflow values:

Negative value:

<i>MAn char-string</i>	Prints at position <i>n</i>
<i>MF char-string</i>	Prints immediately left of a right-justified value or immediately right of a left-justified value
<i>MP char-string</i>	Prints immediately right of a value

Positive value:

<i>PAn char-string</i>	Prints at position <i>n</i>
<i>PF char-string</i>	Prints immediately left of a value
<i>PP char-string</i>	Prints immediately right of a value

Zero value:

<i>ZAn char-string</i>	Prints at position <i>n</i>
<i>ZF char-string</i>	Prints immediately left of a right-justified value or immediately right of a left-justified value
<i>ZP char-string</i>	Prints immediately right of a value

Overflow value:

<i>OAn char-string</i>	Prints at position <i>n</i> .
------------------------	-------------------------------

- These are examples of edit descriptors with decorations:

Format	Item Value	Printed Item
"[MF<','MP'>',ZPP' '] F12.2"	1000.00	1,000.00
"[MF<','MP'>',ZPP' '] F12.2"	-1000.00	<1,000.00>
"[MA1'CR',MPF'\$'] F12.2"	1000.00	\$1,000.00
"[MA1'CR',MPF'\$'] F12.2"	-100.00	CR \$100.00
"[MA1'CR',MPF'\$'] F12.2"	0.00	0.00
"[OA1'**overflow**'] F12.2"	1000000.00	1000000.00
"[OA1'**overflow**'] F12.2"	1000000000.00	**overflow**
"[ZPA2+'] I8"	-10	10
"[ZPA2+'] I8"	100	+ 100
"[ZPA2+'] I8"	0	+0

ZPP specifies that if the value is zero or positive (ZP), the print location is set immediately to the right of the value (P).

MPF specifies that if the value is negative or positive (MP), the print location is set immediately to the left of the value (F).

AS DATE/TIME Clause

The AS DATE/TIME clause specifies a format for printing a date and time in the BREAK FOOTING, BREAK TITLE, DETAIL, PAGE FOOTING, PAGE TITLE, REPORT FOOTING, and REPORT TITLE report writer commands.

Print items printed with AS DATE/TIME must be in Julian timestamp format. To convert dates and times in other formats to Julian timestamps, use the COMPUTE_TIMESTAMP function.

AS	{	DATE	{	*	}	}	[IN	LCT]
		"date-string"	}		}					
	{	TIME	{	*	}	}	[IN	GMT]
		"time-string"	}		}					

*

specifies the default format, which is the current setting of the DATE_FORMAT style option (for a date) and the current setting of the TIME_FORMAT style option (for a time).

"date-string"

is a string that specifies how to format a date. It can include blanks, commas, hyphens, periods, slashes, and these keywords:

- M[n] Specifies a month
- D[n] Specifies a day
- Y[n] Specifies a year
- A[n] Spells out the month or day. If you specify An, only n bytes are displayed.
- B[n] Suppresses leading zeros
- O[n] Spells out the number of the day. If you specify On, only n bytes are displayed.
- n An integer that specifies the minimum number of bytes (1-3) or digits (1-4) to print. The default value of n is 1.

For example, the date formats in this list cause printing of one of the corresponding values:

Format	Values
MA	January, February, ... December
MA3	Jan, Feb, ... Dec
M2	01, 02, ... 12
M	1, 2, ... 12
MB2	1, 2, ... 12

Format	Values
DA	Monday, Tuesday, ... Sunday
DA3	Mon, Tue, ... Sun
D2	01, 02, ... 31
DB2	1, 2, ... 31
D3	001, 002, ... 366
DB3	1, 2, ... 366
DOB2	1st, 2nd, ... 31st
DAO	First, Second, ... Thirty-First
Y2	00, 01, ... 86, 87, 88, ...
YB2	0, 1, ... 86, 87, 88, ...
Y4	1900, 1901, ... 1986, 1987, 1988, ... 2000, 2001, 2002, 2003, 2004, ...

"time-string"

is a string that specifies the method format a time value. It can include blanks, commas, decimal points, hyphens, periods, slashes, alphanumeric characters embedded in the time value enclosed in apostrophes and these keywords:

H[n]	Specifies an hour
M[n]	Specifies a minute
S[n]	Specifies a second
C[n]	Specifies hundredths of a second
T[n]	Specifies thousandths of a second
P[n]	Expresses the hour as modulo 12 with AM or PM
B[n]	Suppresses leading zeros
n	An integer that specifies the minimum number of digits to print. It can be 1 or 2 for all keywords but T, and 1 through 3 for T. The default is 1.

For example, the time formats in these list cause printing of one of the corresponding values:

Format	Values
HB	1, 2, ... 24
HP2	01, 02, ... 12AM or PM
HPB2	1, 2, ..., 12AM or PM
M2	00, 01, ... 59
MB2	0, 1, ... 59
S2	00, 01, ... 59
SB2	0, 1, ... 59

Format	Values
C2	00, 01, ... 99
CB2	0, 1, ... 99
T3	000, 001, ... 999
TB3	0, 1, ... 999

IN LCT

specifies that the value of the item in the database is encoded as local civil time.

IN GMT

specifies that the value of the item in the database is encoded as Greenwich mean time, the default. The value formatted for display is always represented as local civil time.

The range of timestamps for IN GMT starts at 0000/1/1:12:00:00 and ends at 9999/12/31:12:00:00. This allows for the maximum of 12 hours difference between GMT and LCT so that the LCT value falls within 0000/1/1:00:00:00 and 9999/12/31/23:59:59.999999.

Examples—AS DATE/TIME

- This AS clause prints a date in the format March 15, 2004:

```
AS DATE "MA DB2, Y4"
```

- This clause prints a time value in the format 20:03:45:004:

```
AS TIME "HB2:M2:S2:C3"
```

- This clause specifies a date format but uses the default time format:

```
AS DATE "MA DB2, Y4" TIME *
```

ASCII Character Set

The ASCII character set is a subset of the nine single-byte ISO character sets (ISO 8859/1 through ISO 8859/9) that are used by SQL.

[Table A-6](#) is a list of the characters in the ASCII character set along with their internal representations and meanings.

Table A-6. ASCII Character Set (page 1 of 5)

Octal						
Char	Left	Right	Hex	Dec	Meaning	Ordinal
NUL	000000	000000	00	0	Null	1
SOH	000400	000001	01	1	Start of heading	2
STX	001000	000002	02	2	Start of text	3
ETX	001400	000003	03	3	End of text	4
EOT	002000	000004	04	4	End of transmission	5
ENQ	002400	000005	05	5	Enquiry	6
ACK	003000	000006	06	6	Acknowledge	7
BEL	003400	000007	07	7	Bell	8
BS	004000	000010	8	8	Backspace	9
HT	004400	000011	9	9	Horizontal tabulation	10
LF	005000	000012	A	10	Line feed	11
VT	005400	000013	B	11	Vertical tabulation	12
FF	006000	000014	C	12	Form feed	13
CR	006400	000015	D	13	Carriage return	14
SO	007000	000016	E	14	Shift out	15
SI	007400	000017	F	15	Shift in	16
DLE	010000	000020	10	16	Data link escape	17
DC1	010400	000021	11	17	Device control 1	18
DC2	011000	000022	12	18	Device control 2	19
DC3	011400	000023	13	19	Device control 3	20
DC4	012000	000024	14	20	Device control 4	21
NAK	012400	000025	15	21	Negative acknowledge	22
SYN	013000	000026	16	22	Synchronous idle	23
ETB	013400	000027	17	23	End of transmission block	24
CAN	014000	000030	18	24	Cancel	25
EM	014400	000031	19	25	End of medium	26
SUB	015000	000032	1A	26	Substitute	27

Table A-6. ASCII Character Set (page 2 of 5)

Octal						
Char	Left	Right	Hex	Dec	Meaning	Ordinal
ESC	015400	000033	1B	27	Escape	28
FS	016000	000034	1C	28	File separator	29
GS	016400	000035	1D	29	Group separator	30
RS	017000	000036	1E	30	Record separator	31
US	017400	000037	1F	31	Unit separator	32
SP	020000	000040	20	32	Space	33
!	020400	000041	21	33	Exclamation point	34
"	021000	000042	22	34	Quotation mark	35
#	021400	000043	23	35	Number sign	36
\$	022000	000044	24	36	Dollar sign	37
%	022400	000045	25	37	Percent sign	38
&	023000	000046	26	38	Ampersand	39
'	023400	000047	27	39	Apostrophe	40
(024000	000050	28	40	Opening parenthesis	41
)	024400	000051	29	41	Closing parenthesis	42
*	025000	000052	2A	42	Asterisk	43
+	025400	000053	2B	43	Plus	44
,	026000	000054	2C	44	Comma	45
-	026400	000055	2D	45	Hyphen (minus)	46
.	027000	000056	2E	46	Period (decimal point)	47
/	027400	000057	2F	47	Right slash	48
0	030000	000060	30	48	Zero	49
1	030400	000061	31	49	One	50
2	031000	000062	32	50	Two	51
3	031400	000063	33	51	Three	52
4	032000	000064	34	52	Four	53
5	032400	000065	35	53	Five	54
6	033000	000066	36	54	Six	55
7	033400	000067	37	55	Seven	56
8	034000	000070	38	56	Eight	57
9	034400	000071	39	57	Nine	58
:	035000	000072	3A	58	Colon	59
;	035400	000073	3B	59	Semicolon	60

Table A-6. ASCII Character Set (page 3 of 5)

Octal						
Char	Left	Right	Hex	Dec	Meaning	Ordinal
<	036000	000074	3C	60	Less than	61
=	036400	000075	3D	61	Equals	62
>	037000	000076	3E	62	Greater than	63
?	037400	000077	3F	63	Question mark	64
@	040000	000100	40	64	Commercial at sign	65
A	040400	000101	41	65	Uppercase A	66
B	041000	000102	42	66	Uppercase B	67
C	041400	000103	43	67	Uppercase C	68
D	042000	000104	44	68	Uppercase D	69
E	042400	000105	45	69	Uppercase E	70
F	043000	000106	46	70	Uppercase F	71
G	043400	000107	47	71	Uppercase G	72
H	044000	000110	48	72	Uppercase H	73
I	044400	000111	49	73	Uppercase I	74
J	045000	000112	4A	74	Uppercase J	75
K	045400	000113	4B	75	Uppercase K	76
L	046000	000114	4C	76	Uppercase L	77
M	046400	000115	4D	77	Uppercase M	78
N	047000	000116	4E	78	Uppercase N	79
O	047400	000117	4F	79	Uppercase O	80
P	050000	000120	50	80	Uppercase P	81
Q	050400	000121	51	81	Uppercase Q	82
R	051000	000122	52	82	Uppercase R	83
S	051400	000123	53	83	Uppercase S	84
O	047400	000117	4F	79	Uppercase O	80
P	050000	000120	50	80	Uppercase P	81
Q	050400	000121	51	81	Uppercase Q	82
R	051000	000122	52	82	Uppercase R	83
S	051400	000123	53	83	Uppercase S	84
T	052000	000124	54	84	Uppercase T	85
U	052400	000125	55	85	Uppercase U	86
V	053000	000126	56	86	Uppercase V	87
W	053400	000127	57	87	Uppercase W	88

Table A-6. ASCII Character Set (page 4 of 5)

Octal						
Char	Left	Right	Hex	Dec	Meaning	Ordinal
X	054000	000130	58	88	Uppercase X	89
Y	054400	000131	59	89	Uppercase Y	90
Z	055000	000132	5A	90	Uppercase Z	91
[055400	000133	5B	91	Opening bracket	92
\	056000	000134	5C	92	Back slash	93
]	056400	000135	5D	93	Closing bracket	94
^	057000	000136	5E	94	Circumflex	95
_	057400	000137	5F	95	Underscore	96
**	060000	000140	60	96	“grave” accent	97
a	060400	000141	61	97	Lowercase a	98
b	061000	000142	62	98	Lowercase b	99
c	061400	000143	63	99	Lowercase c	100
d	062000	000144	64	100	Lowercase d	101
e	062400	000145	65	101	Lowercase e	102
f	063000	000146	66	102	Lowercase f	103
g	063400	000147	67	103	Lowercase g	104
h	064000	000150	68	104	Lowercase h	105
i	064400	000151	69	105	Lowercase i	106
j	065000	000152	6A	106	Lowercase j	107
k	065400	000153	6B	107	Lowercase k	108
l	066000	000154	6C	108	Lowercase l	109
m	066400	000155	6D	109	Lowercase m	110
n	067000	000156	6E	110	Lowercase n	111
o	067400	000157	6F	111	Lowercase o	112
p	070000	000160	70	112	Lowercase p	113
q	070400	000161	71	113	Lowercase q	114
r	071000	000162	72	114	Lowercase r	115
s	071400	000163	73	115	Lowercase s	116
t	072000	000164	74	116	Lowercase t	117
u	072400	000165	75	117	Lowercase u	118
v	073000	000166	76	118	Lowercase v	119
w	073400	000167	77	119	Lowercase w	120
x	074000	000170	78	120	Lowercase x	121

Table A-6. ASCII Character Set (page 5 of 5)

Octal						
Char	Left	Right	Hex	Dec	Meaning	Ordinal
y	074400	000171	79	121	Lowercase y	122
z	075000	000172	7A	122	Lowercase z	123
{	075400	000173	7B	123	Opening brace	124
	076000	000174	7C	124	Vertical line	125
}	076400	000175	7D	125	Closing brace	126
~	077000	000176	7E	126	Tilde	127
DEL	077400	000177	7F	127	Delete	128

** The “grave” accent character is not printable

AUDIT File Attribute

AUDIT is a Guardian file attribute that determines whether or not a table is audited by TMF. AUDIT applies to key-sequenced, relative, and entry-sequenced tables.

Audited tables are protected by TMF, but you must perform all input and output on an audited table (except for queries with BROWSE access) within a TMF transaction. Nonaudited tables can be accessed without a transaction in progress but are not protected by TMF. If you turn off AUDIT, TMF online dumps become invalid and cannot be used for subsequent recovery operations.

For more information, see [TMF Transactions](#) on page T-6.

```
{ AUDIT | NO AUDIT }
```

The table default is AUDIT.

Considerations—AUDIT

- An index always has the same AUDIT attribute as its underlying table. If you change the AUDIT attribute for a table, SQL automatically changes the AUDIT attribute for its indexes.
- A protection view has the same AUDIT value as the underlying table.

For a shorthand view, the AUDIT value is:

- AUDIT if all referenced tables and views are audited.
- NO AUDIT if all referenced tables and views are nonaudited.
- Mixed if any of the referenced views are mixed or if at least one of the referenced tables or views is audited and at least one is nonaudited.

The value stored in the VIEWS table is Y (for AUDIT), N (for NO AUDIT), or M (for mixed).

- For a given table, all partitions are audited, or all partitions are nonaudited.
- If you alter the AUDIT file attribute for a table, SQL automatically sets the BUFFERED file attribute for that table (but not for its dependent indexes):
 - If you specify AUDIT, SQL also sets BUFFERED.
 - If you specify NO AUDIT, SQL also sets NO BUFFERED.

To override the automatic setting, explicitly specify the BUFFERED file attribute in the ALTER TABLE statement that changes the AUDIT attribute.

Altering the BUFFERED attribute does not affect the AUDIT attribute.

AUDITCOMPRESS File Attribute

AUDITCOMPRESS is a Guardian file attribute that controls the TMF audit records are compressed. AUDITCOMPRESS applies to audited key-sequenced, relative, and entry-sequenced tables and to indexes.

Compressed audit records omit unchanged columns from the before and after images of updated rows. Uncompressed audit records allow you to read complete rows in the audit trail but require more space.

```
{ AUDITCOMPRESS | NO AUDITCOMPRESS }
```

The table default is AUDITCOMPRESS.

The index default is the table value at index creation.

Considerations—AUDITCOMPRESS

- AUDITCOMPRESS can save system resources, so you should allow tables and indexes to default to AUDITCOMPRESS, unless you have a specific requirement to read audit trail files with complete before and after images.
- Audit records of uncompressed files contain entire before and after images of changed rows. Audit records of compressed files generally contain only changed columns and columns of the primary key. Other columns are occasionally included to improve performance, such as when a single unchanged column physically occurs between several changed columns.
- Programs can read audit trails written with or without audit compression by using TMF audit-reading procedures.
- If AUDITCOMPRESS is on at the start of a CREATE INDEX operation that uses the WITH SHARED ACCESS option, NonStop SQL/MP turns AUDITCOMPRESS off. When finished with the CREATE INDEX operation, SQL turns AUDITCOMPRESS back on. If the CREATE INDEX operation fails, however, SQL

might leave the AUDITCOMPRESS attribute off. You can use the ALTER TABLE statement to turn the attribute back on.

Audited Tables

Audited tables are tables audited by TMF. TMF monitors all transactions against audited tables in preparation for possible transaction backout or TMF recovery operations.

NonStop SQL/MP creates audited tables by default, but you can specify the creation of a nonaudited table (or change an audited table to a nonaudited table) with the AUDIT file attribute for the table.

All SQL/MP catalog tables are audited. In addition, SQL tables must reside on audited volumes, even if the tables themselves are not audited.

For information, see [TMF Transactions](#) on page T-6.

AVG Function

AVG is a function that computes the average of a set of numbers.

The data type of the result depends on the data type of the argument. If the argument is an exact numeric type, the result is LARGEINT. If the argument is FLOAT, REAL, or DOUBLE PRECISION type, the result is DOUBLE PRECISION.

The scale of the result is the same as the scale of the argument. If the argument has no scale, the result is truncated.

<pre>AVG { ([ALL] expression) } { (DISTINCT column) }</pre>
--

[ALL] *expression*

specifies a numeric or INTERVAL expression that indicates the set of values to average.

The expression must include a value from each row of the result table (that is, at least one column from the result table), and cannot include the COUNT, MAX, MIN, or SUM functions, or another AVG function. For example,

```
AVG ( SALARY )
AVG ( PARTCOST * QTY_ORDERED )
```

ALL is an optional keyword that does not change the meaning of the clause. Unless you use the DISTINCT clause, SQL uses all rows whether or not you specify ALL.

`DISTINCT column`

specifies a set of distinct column values from each row of the result table to average. The column cannot be a column from a view that corresponds to an expression in the view definition.

If you specify `DISTINCT` in more than one `AVG` function in the same statement, the functions must reference the same column.

Considerations—AVG

- `AVG` is evaluated after eliminating all null values from the aggregate set. If the result set is empty, `AVG` returns a null.
- A host variable that receives the result of `AVG` must have an indicator variable to handle a possible null value. (For more information about using indicator variables, see the SQL/MP programming manual for your host language.)

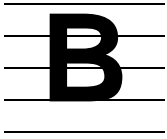
Examples—AVG

- This `SELECT` statement returns the average salary from the `SALARY` column of the `PERSNL.EMPLOYEE` table:

```
>> SELECT AVG (SALARY) FROM PERSNL.EMPLOYEE;
(EXPR)
-----
                48784.65
--- 1 row(s) selected.
```

- This `SELECT` statement returns the average unique salary from the `SALARY` column of the `PERSNL.EMPLOYEE` table:

```
>> SELECT AVG (DISTINCT SALARY) FROM PERSNL.EMPLOYEE;
(EXPR)
-----
                52664.21
--- 1 row(s) selected.
```

BACKUP Utility

BACKUP is a Guardian utility program, executed from TACL, that copies Guardian files and SQL objects from disk to magnetic tape. (A complementary utility program, RESTORE, copies Guardian files and SQL objects from the magnetic tape to the disk.)

For more information about BACKUP, see the *Guardian Disk and Tape Utilities Reference Manual*.

BASETABS Table

The BASETABS table is a catalog table that describes attributes that apply to tables, but not to views. [Table B-1](#) describes the contents of the BASETABS table.

Table B-1. The BASETABS Table

Column Name	Data Type	Description
1 TABLENAME	CHAR (34)	Name of table (primary key)
2 CONSTRAINTS	CHAR (1)	Y if table has constraints N if no constraints
3 ALLINDEXESHERE	CHAR (1)	Y if all indexes for table are described in the same catalog N if some indexes are described in other catalogs
4 FILENAME	CHAR (34)	Name of file for table
5 STATISTICSTIME	LARGEINT SIGNED	Time when statistics for table were last updated, in GMT.
6 ROWCOUNT	LARGEINT SIGNED	Number of rows in table or partition of table identified by TABLENAME (updated by UPDATE STATISTICS)
7 ROWSIZE	SMALLINT UNSIGNED	Maximum byte length of a row of the table on disk (packed record length)
8 VALIDDEF	CHAR (1)	Y if file has a valid definition, correct file label, and catalog entries N if not
9 VALIDDATA	CHAR (1)	Y if data in table is consistent with data in indexes and satisfies constraints on table N if not

The BASETABS table was created in version 1, and there have been no changes in subsequent versions.

If a table is partitioned, the BASETABS table has one entry for each partition. Guardian names in the BASETABS table are fully qualified and use uppercase characters.

BEGIN DECLARE SECTION Directive

BEGIN DECLARE SECTION is a host program directive that starts a host program Declare Section for declaring host variables to use in SQL statements.

Guidelines for the use of Declare Sections vary with the host language. For more information, see the SQL/MP programming manual for your host language.

```
BEGIN DECLARE SECTION
```

Examples—BEGIN DECLARE SECTION

- This example shows a portion of a COBOL Working Storage Section that includes an SQL Declare Section. Variables declared within the Working Storage Section but outside the SQL Declare Section cannot be used in SQL statements.

```
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
    01 SUPPLIER          PIC 999.
    01 ORDERS OCCURS     10 TIMES.
        05 ORDERNUM      PIC 999 COMP.
        05 DT             DATE.
EXEC SQL END DECLARE SECTION    END-EXEC.
...
```

- This example shows a Declare Section that uses an INVOKE directive to declare a table as a record description in a C, Pascal, or TAL program. (A COBOL version would use END-EXEC. in place of each semicolon.)

```
EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL INVOKE SALES.PARTS;
EXEC SQL END DECLARE SECTION;
```

BEGIN WORK Statement

BEGIN WORK is a transaction control statement that starts a user-defined TMF transaction.

A user-defined TMF transaction groups a set of operations on audited objects and files so that changes made by the operations can be committed (with the COMMIT WORK statement) or rolled back (with the ROLLBACK WORK statement) as a unit. TMF transactions do not protect nonaudited tables. For more information on TMF transactions, see [TMF Transactions](#) on page T-6.

BEGIN WORK

Example—BEGIN WORK

This example uses BEGIN WORK in SQLCI to group three separate statements that update the database into a single TMF transaction:

```
>> VOLUME SALES;
>> BEGIN WORK;
>> INSERT INTO ORDERS VALUES (124, 860323, 860330, 75, 7654);
--- 1 ROW(S) INSERTED.

>> INSERT INTO ODETAIL VALUES (124, 4103, 25000, 2);
--- 1 ROW(S) INSERTED.

>> UPDATE INVENT.PARTLOC SET QTY_ON_HAND = QTY_ON_HAND -2
+>   WHERE PARTNUM = 4103 AND LOC_CODE = "K43";
--- 1 ROW(S) UPDATED.
>> COMMIT WORK;
```

BETWEEN Predicate

BETWEEN is a predicate that determines whether a value is within a range of values.

```

row-value-spec [NOT] BETWEEN row-value-spec AND
    row-value-spec

row-value-spec is:
    { expression[ ,expression] ... }
    { (expression[ ,expression] ... ) }

```

Considerations—BETWEEN

- BETWEEN is a comparison predicate. For more information on general rules for comparisons and specific information about comparing character data (including character data associated with collations), numeric data, date-time data, and interval data, see [Comparison Predicate](#) on page C-58.

- If you specify *expr1* BETWEEN *expr2* AND *expr3*, the predicate is true if this condition is true:

```
expr1 >= expr2 AND expr1 <= expr3
```

- The three *row-value-specs* must contain the same number of expressions.
- The data type of the result of an expression in one *row-value-spec* must be compatible with the data types of the results of the two expressions in the same ordinal position in the others.
- If you specify NOT, the predicate is true if this condition is true:

```
( expr2 < expr1 OR expr1 > expr3 )
```

Specify NOT:

```
expr1 NOT BETWEEN expr2 AND expr3
```

- For a clause that specifies a column in a key BETWEEN *expr1* and *expr2*, *expr2* must be greater than *expr1* even if the column is defined as DESCENDING.

Examples—BETWEEN

- This example finds those items for which the total price of the units in inventory is in the range \$1,000 to \$10,000:

```

QTY_ON_HAND * PRICE
    BETWEEN 1000.00 AND 10000.00

```

- This example finds those items for which the part cost is less than \$5 or more than \$800:

```
PARTCOST NOT BETWEEN 5.00 AND 800.00
```

- This example finds those names between Jody Selby and Gene Wright. The name Barbara Swift meets the criteria; the name Mike Wright does not.

```
LAST_NAME, FIRST_NAME BETWEEN  
  "SELBY", "JODY" AND  
  "WRIGHT", "GENE"
```

BLOCKSIZE File Attribute

BLOCKSIZE is a Guardian file attribute that specifies the number of bytes in a block. BLOCKSIZE applies to key-sequenced, relative, and entry-sequenced tables, and to indexes.

BLOCKSIZE <i>num-bytes</i>

The default is BLOCKSIZE 4096.

num-bytes

is an integer that specifies the number of bytes in a block.

Block size can be 512, 1024, 2048, or 4096 bytes. If you specify a different block size, SQL uses the next-higher block size and issues a warning. If you specify a block size greater than 4096 for a CREATE TABLE or CREATE INDEX statement, SQL issues an error.

Considerations—BLOCKSIZE

Recommendations for block size

For sequential processing, use the largest block size.

For key-sequenced tables, avoid small block sizes (less than 4096 bytes) if the size of the table causes the number of index levels to increase over the number required for 4096-byte blocks. Additional index levels can reduce online performance.

If your application uses a table or an index to process data solely with random access, choose one of the smaller block sizes for that table or index. Note, however, that small block sizes require more disk space for the same number of rows than large block sizes.

BREAK FOOTING Command

BREAK FOOTING is an SQLCI report writer command that specifies the text at the end of a group of break column values. You can use BREAK FOOTING only from the select-in-progress prompt, not from the SQLCI prompt.

```
BREAK FOOTING break-column (print-list) [CENTER];

print-list is:

    print-item [, print-item] ...
```

break-column

identifies a break column (a column named in a BREAK ON command). It can be a column name, an alias, a detail alias, or COL *number* (which specifies the position of the column in the select list).

print-item

specifies the contents and format of items to print when the value in the break column changes. *print-item* is the same as described under the DETAIL command except that it cannot include the HEADING, NOHEAD, or NAME clauses. For more information, see [DETAIL Command](#) on page D-47 or [Print Item](#) on page P-28.

If *print-item* is a column identifier, the column value will be the one from the last select row in the group.

CENTER

centers each line of the break footing between the left and right margins. If you omit CENTER, the break footing is positioned at the left margin.

Considerations—BREAK FOOTING

- If you define break footings, you must enter a BREAK ON command that includes the break column identifier in each BREAK FOOTING command before you list any output.
- Only one BREAK FOOTING command is in effect for each break column. If you enter another BREAK FOOTING command for the same break column, the most recent command replaces the previous BREAK FOOTING command for that column.
- The print list you specify in a BREAK FOOTING command is a logical line, although (depending on margin settings, device widths, and use of the SKIP clause) it might print on more than one physical line. A logical line is limited to 4072

bytes, including the field widths of all print items and the number of spaces between items.

Example—BREAK FOOTING

This example uses break columns to format a report. The report includes a break footing.

```
>> SELECT D.DEPTNUM, DEPTNAME, EMPNUM, JOBCODE, LOCATION
+>   FROM PERSNL.EMPLOYEE E, PERSNL.DEPT D
+>   WHERE E.DEPTNUM = D.DEPTNUM
+>   ORDER BY D.DEPTNUM;
S> BREAK ON COL 1, COL 2;
S> DETAIL D.DEPTNUM, DEPTNAME, EMPNUM, JOBCODE;
S> BREAK FOOTING D.DEPTNUM ("Location:", SPACE 3,
+>   LOCATION AS A20);
S> LIST NEXT 15;
```

DEPTNUM	DEPTNAME	EMPNUM	JOBCODE
-----	-----	-----	-----
1000	FINANCE	23	100
		202	500
		208	900
		210	500
		214	500
Location:	CHICAGO		
1500	PERSONNEL	209	900
		211	600
		212	600
		213	100
Location:	CHICAGO		
2000	INVENTORY	32	100
		219	250
		230	200
		233	250
		321	900
Location:	LOS ANGELES		

BREAK ON Command

BREAK ON is an SQLCI report writer command that groups detail lines together by the value of a specified column. You can use BREAK ON only from the select-in-progress prompt, not from the standard SQLCI prompt.

```
BREAK ON col [SUPPRESS] [, col [SUPPRESS]]... ;
           [NOSUPPRESS] [           [NOSUPPRESS]]
```

The default is SUPPRESS.

col

identifies a column of the select list or a named detail column, the value of which is used to group lines. *col* can be a column name, an alias, a detail alias, or COL *number* (which specifies the position of the column in the select list).

A break occurs when the value of *col* changes, even if *col* is not a column in the DETAIL list. The break can cause the printing of a break title, break footing, or subtotal.

You can specify the same column more than once in a BREAK ON command, but report writer sets only one break on the column.

SUPPRESS

specifies printing the break column value only in the first detail line of the group and the first detail line on a page.

NOSUPPRESS

specifies printing the break column value on each detail line.

Considerations—BREAK ON

- Only one BREAK ON command is in effect at any time. When you enter a BREAK ON command, it replaces any existing BREAK ON command and returns you to the beginning of the SELECT output. The new BREAK ON command should include every break column referred to in the existing SUBTOTAL, BREAK TITLE, and BREAK FOOTING commands.
- If you use BREAK ON to specify groups and subgroups, specify the columns in order, from the most inclusive group to the least inclusive group.
- To sort break columns, include an ORDER BY clause in the associated SELECT statement.

Examples—BREAK ON

- This example groups detail lines by job codes within groups of departments. When finished, enter CANCEL at the select-in progress prompt (S>):

```
>> SET LIST_COUNT 0;
>> SELECT LAST_NAME, FIRST_NAME, JOBCODE, DEPTNUM
+>   FROM PERSNL.EMPLOYEE
+>   WHERE SALARY > 20000
+>   ORDER BY DEPTNUM, JOBCODE;
S> BREAK ON DEPTNUM, JOBCODE;
S> LIST NEXT 8;
LAST_NAME          FIRST_NAME          JOBCODE  DEPTNUM
-----
HOWARD              JERRY              100      1000
CLARK               LARRY              500
BARTON              RICHARD
KELLY               JULIA
WHITE               ROBERT             100      1500
SCHNEIDER           JIMMY              600
MITCHELL            JONATHAN
RUDLOFF             THOMAS             100      2000

S> CANCEL;
>>
```

- This example groups detail lines by monthly salary and (within salary groups) by job code and department number. When finished, enter CANCEL at the select-in progress prompt (S>):

```
>> SET LIST_COUNT 0;
>> SELECT DEPTNUM, JOBCODE, SALARY/12
+>   FROM PERSNL.EMPLOYEE
+>   WHERE SALARY > 20000
+>   ORDER BY 3, 2, 1;
S> BREAK ON COL 3, JOBCODE, DEPTNUM;
S> LIST NEXT 5;
DEPTNUM  JOBCODE  (EXPR)
-----
3200      300      1833.333333333333
2000      200      2000.000000000000
4000      900      2000.075000000000
3200      900      2083.333333333333
1000      500      2083.395833333333

S> CANCEL;
>>
```

BREAK TITLE Command

BREAK TITLE is an SQLCI report writer command that specifies the text at the beginning of a group of break column values. You can use BREAK TITLE only from the select-in-progress prompt, not from the SQLCI prompt.

```
BREAK TITLE break-column (print-list) [CENTER];

print-list is:
    print-item> [, print-item] ...
```

break-column

identifies a break column (a column named in a BREAK ON command). It can be a column name, an alias, a detail alias, or COL *number* (which specifies the position of the column in the select list).

print-item

specifies the contents and format of items to print as the break title. *print-item* is the same as described under DETAIL, except that it cannot include the HEADING, NOHEAD, or NAME clauses.

If *print-item* is a column identifier, the column value used is the one from the first row in the new group.

CENTER

centers each line of the break title between the left and right margins. If you omit CENTER, the break title is positioned at the left margin.

Considerations—BREAK TITLE

- If you define break titles, you must enter a BREAK ON command that includes the break column identifier in each BREAK TITLE command before listing any output.
- Only one BREAK TITLE command is in effect for each break column. If you enter another BREAK TITLE command for the same break column, the most recent command replaces the previous BREAK TITLE command for that column.
- The print list you specify in a BREAK TITLE command is a logical line, although (depending on margin settings, device widths, and use of the SKIP clause) it might print on more than one physical line. A logical line is limited to 4072 bytes, including the field widths of all print items and the number of spaces between items.

Example—BREAK TITLE

These commands select data and identify DEPTNUM as a break column:

```
>> SELECT D.DEPTNUM, DEPTNAME, EMPNUM, JOBCODE
+> FROM PERSNL.EMPLOYEE E, PERSNL.DEPT D
+> WHERE E.DEPTNUM = D.DEPTNUM
+> ORDER BY D.DEPTNUM;
S> DETAIL D.DEPTNUM, EMPNUM, JOBCODE;
S> BREAK ON COL 1;
```

Before the first detail line is printed, and each time the value of DEPTNUM changes, the break title defined in this command appears in the report:

```
S> BREAK TITLE D.DEPTNUM
+> ( CONCAT (DEPTNAME STRIP, " Department") );
S> LIST NEXT 8;
DEPTNUM  EMPNUM  JOBCODE
-----  -
FINANCE Department
    1000      23      100
           202      500
           208      900
           210      500
           214      500
PERSONNEL Department
    1500     209      900
           211      600
           212      600
```

BUFFERED File Attribute

BUFFERED is a Guardian file attribute that specifies whether to buffer writes to a disk file. BUFFERED applies to key-sequenced, relative, entry-sequenced tables, and to indexes.

{ BUFFERED NO BUFFERED }

The default for an audited table is BUFFERED. The default for a nonaudited table is NO BUFFERED. The index default is the table value at index creation.

Considerations—BUFFERED

- All retrieved rows are stored in cache memory temporarily. In a NO BUFFERED file, a block that contains an updated row is written to disk immediately. In a BUFFERED file, a block that contains an updated row is not written to disk until a system event (such as the need for more cache memory) triggers a write operation. An updated block can change more than once without the need for writing each change to disk as the change occurs.
- Buffering can improve transaction times by reducing the number of writes required and by deferring writes so that the disk process can write a string of blocks in a

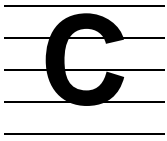
single I/O operation. However, buffering can cause loss of data on nonaudited files if a failure occurs while updated rows are stored in cache memory but not yet written to disk.

Audited files should always be buffered, because auditing itself protects against loss of data. (Specifying NO BUFFERED for an audited file would unnecessarily reduce performance.)

- If you alter the AUDIT file attribute for a table, SQL automatically sets the BUFFERED file attribute for that table (but not for its dependent indexes):
 - If you specify AUDIT, SQL also sets BUFFERED.
 - If you specify NO AUDIT, SQL also sets NO BUFFERED.

To override the automatic setting, explicitly specify the BUFFERED file attribute in the ALTER TABLE statement that changes the AUDIT attribute.

Altering the BUFFERED attribute does not affect the AUDIT attribute.



CANCEL Command

CANCEL is an SQLCI report writer command that cancels the current SELECT and returns to the standard SQLCI prompt. Selected rows and current report formatting commands become unavailable, except through the FC command.

```
CANCEL ;
```

Consideration—CANCEL

- CANCEL does not delete reports. You can still display or save a report after using CANCEL:

SHOW REPORT *; Displays a report

SAVE REPORT * TO *file*; Saves a report

Example—CANCEL

This example lists the first five rows of SELECT output and cancels the SELECT statement:

```
>> SET LIST_COUNT 5;
>> SELECT * FROM SALES.ODETAIL;
ORDERNUM  PARTNUM  UNIT_PRICE  QTY_ORDERED
-----
100124    4103      25000.00    2
100210     244       3500.00    2
100210    2001      1100.00    3
100210    2403       620.00    3
100210    5100       150.00    6
S> CANCEL;
>>
```

CASE Expression

The CASE expression is a conditional expression. SQL evaluates the conditions in the CASE expression and sets the CASE expression to a value based on the condition that is true. If none of the search conditions are true, SQL sets the CASE expression to the value specified in the ELSE clause or, if ELSE is not specified, SQL sets the CASE expression to NULL.

```

CASE
  WHEN search-condition-1 THEN { result-1 | NULL }
  WHEN search-condition-2 THEN { result-2 | NULL }
  WHEN search-condition-3 THEN { result-3 | NULL }
  . . .
  WHEN search-condition-n THEN { result-n | NULL }
  [ { ELSE result-x | NULL } ]
END

or

CASE target-value
  WHEN value-1 THEN { result-1 | NULL }
  WHEN value-2 THEN { result-2 | NULL }
  WHEN value-3 THEN { result-3 | NULL }
  . . .
  WHEN value-n THEN { result-n | NULL }
  [ { ELSE result-x | NULL } ]
END

```

search-condition-1 through *search-condition-n*

specifies a condition to test for. If the condition is true, the CASE expression returns the associated *result* value. If no *search-condition* is true, the CASE expression returns the value of the ELSE clause, or NULL if ELSE is not specified.

result-1 through *result-n*

specifies the result value (or NULL) associated with a specific search condition. All the results specified in the CASE expression should have the same or comparable data types.

result-x

specifies the value returned if none of the search conditions are true. If the ELSE *result-x* clause is not specified, the CASE expression returns NULL if none of the search conditions are true. The data type of *result-x* should be the same or comparable to those of *result-1* through *result-n*.

target-value

if present, specifies a value or expression for which a result is returned. When you specify *target-value*, you use an abbreviated form of the CASE expression, typically used for value comparisons. The data type of each *value-n* in the statement should be comparable to the data type of *target-value*. In addition, the collation of each *value-n* should be the same or comparable to the collation of *target-value*.

value-1 through *value-n*

specifies a value associated with *result-n*. If the value matches *target-value*, the CASE expression returns the associated result.

Considerations—CASE Expression

- The data type of the CASE expression depends on the data type of the result expressions. If all the results have the same data type, the CASE expression adopts that data type. If all the results are untyped, the CASE expression has a CHAR (256) data type. If all the results are comparable but not identical data types, the CASE expression adopts the data type, which is superior among the results associated with the THEN clause or the ELSE clause, if present.
- If you plan to use the value of the CASE expression in a comparison (as in a WHERE clause), the character set and collation associated with the CASE expression should be the same or comparable to those associated with the comparison expression.
- If none of the *search-conditions* are true, the value of the CASE expression is the result associated with the ELSE clause, if present.
- At least one specified result should have a nonnull value.
- A SELECT clause cannot appear as part of a *search-condition* if the CASE expression is part of the select list in a SELECT statement. For example, this statement is not valid:

```
SELECT CASE WHEN a IN (SELECT b FROM T1) THEN 1
           WHEN b IN (SELECT c FROM T1) THEN 2
           ELSE 3
        END
FROM table1;
```

You can, however, use the CASE expression in the WHERE clause of a SELECT statement.

- Use of the abbreviated form of the CASE expression, CASE *value...*, is equivalent to using:

```
CASE
  WHEN target-value = value-1 THEN result-1
  WHEN target-value = value-2 THEN result-2
```

```

    WHEN target-value = value-3 THEN result-3
    ...
    WHEN target-value = value-n THEN result-n
    ELSE value-x
END

```

Examples—CASE Expression

- This example decodes `movie_type` and returns NULL if `movie_type` does not match any of the listed values:

```

SELECT movie_name,
       CASE movie_type
         WHEN 1 THEN "Horror"
         WHEN 2 THEN "Comedy"
         WHEN 3 THEN "Drama"
         ELSE NULL
       END
FROM movies;

```

- This example returns `last_name`, `first_name`, and a value based on salary that depends on the value of `employee.dept_num`:

```

SELECT last_name, first_name,
       CASE
         WHEN dept_num = 9000 THEN salary * 1.10
         WHEN dept_num = 1000 THEN salary * 1.12
         ELSE salary
       END
FROM employee;

```

CAST Function

The CAST expression converts data to the data type you specify.

```
CAST ( expression AS data-type )
```

expression

specifies the operand to convert to the data type *data-type*.

If the operand is an expression, *data-type* depends on the data type of *expression* and follows the rules outlined under [Valid Considerations—CAST](#) on page C-5.

If the value of the expression is null, the result of CAST is NULL, regardless of the data type you specify.

data-type

specifies a data type to associate with the operand of CAST. For more information, see [Data Types](#) on page D-1.

When casting data to a CHAR or VARCHAR data type, the resulting data value is left justified. Otherwise, the resulting data value is right justified. Further, when you are casting to a CHAR or VARCHAR data type, you must specify the length of the target value.

Valid Considerations—CAST

You can convert:

- An exact or approximate numeric value to any other numeric data type.
- An exact or approximate numeric value to any character string data type.
- An exact numeric value to either a single-field year-month or day-time interval, such as INTERVAL '30' DAY.
- A character string to any other data type, with this restriction:

The contents of the character string to be converted must be consistent in meaning with the data type of the result. For example, if you are converting to DATE, the contents of the character string must be 10 characters, consisting of the year, the hyphen, the month, another hyphen, and the day.

- A date value to a character string or to a TIMESTAMP (NonStop SQL/MP fills in the time part with 00:00:00.00).
- A time value to a character string or to a TIMESTAMP (NonStop SQL/MP fills in the date part with the current date).
- A timestamp value to a character string, a DATE, a TIME, or another DATETIME with different fractional seconds precision.
- A year-month interval value to a character string, an exact numeric, or to another year-month INTERVAL with a different start field precision.
- A day-time interval value to a character string, an exact numeric, or to another day-time INTERVAL with a different start field precision.

You cannot perform the following conversions between the datatypes:

- NUMERIC to DATETIME, INTERVAL to DATETIME
- DATETIME to NUMERIC, DATETIME to INTERVAL
- NUMERIC to INTERVAL, INTERVAL to NUMERIC
- INTERVAL YEAR TO MONTH to INTERVAL DAY TO FRACTION
- INTERVAL DAY TO FRACTION to INTERVAL YEAR TO MONTH
- INTERVAL to FLOAT, FLOAT to INTERVAL
- INTERVAL to character (other than fixed length char)
- Character (other than fixed length char) to INTERVAL

- Multibyte datatype to any non-character datatype
- Any non-character datatype to multibyte datatype

Examples—CAST

- The PROJECT table contains a column START_DATE of data type DATE and a column SHIP_TIMESTAMP of data type TIMESTAMP.

Use CAST to return the number of days for completion of a project:

```
SELECT projdesc, start_date, ship_timestamp,
       (CAST (ship_timestamp AS DATE) - start_date) DAY
FROM persnl.project;
```

PROJDESC	START_DATE	SHIP_TIMESTAMP	(EXPR)
SALT LAKE CITY	1996-04-10	1996-04-21 08:15:00.000000	11
ROSS PRODUCTS	1996-06-10	1996-07-21 08:30:00.000000	41
MONTANA TOOLS	1996-10-10	1996-12-21 09:00:00.000000	72
AHAUS TOOL	1996-08-21	1996-10-21 08:10:00.000000	61
THE WORKS	1996-09-21	1996-10-21 10:15:00.000000	30

--- 5 row(s) selected.

DATE differences can be expressed only in the number of days, the least significant unit of measure for dates. (An interval is either year-month or day-time.) In this example, the result is the same if you express the difference as:

```
CAST (ship_timestamp AS DATE) - start_date
```

You are not required to specify the interval qualifier.

- Suppose that your database includes a log file of user information. This example converts the current timestamp to a character string and concatenates the result to a character literal. Note the length must be specified.

```
INSERT INTO stats.logfile
(user_key, user_info)
VALUES (001, 'User JBrook, executed at ' ||
        CAST (CURRENT_TIMESTAMP AS CHAR(26)));
```

CATALOG Command

CATALOG is an SQLCI command that sets the default catalog in an SQLCI session.

```
CATALOG [ catalog ] ;
```

catalog

is a catalog name (a Guardian subvolume name) with or without a node and volume qualifier. It cannot be a DEFINE.

If you omit *catalog*, SQLCI resets the default to the value of the CATALOG attribute of the =_DEFAULTS DEFINE that SQLCI inherited from the process that started the session. If you omit the node and volume qualifier, SQLCI uses the current defaults.

Considerations—CATALOG

- Whenever the default catalog is empty, SQL uses the default subvolume for the default catalog.
- Unless a CONTROL QUERY BIND NAMES AT EXECUTION directive is in effect when a PREPARE executes, a prepared statement uses the node, volume, and catalog defaults in effect at the time of the PREPARE and is not affected by a change in the default catalog. For more information, see [CONTROL QUERY Directive](#) on page C-74 or [Name Resolution](#) on page N-2.
- CATALOG alters the CATALOG attribute of the =_DEFAULTS DEFINE. Changing the CATALOG attribute with ALTER DEFINE =_DEFAULTS has exactly the same effect as issuing a CATALOG command.

Example—CATALOG

```
CATALOG \SYS1.$VOL1.PERSNL;
```

```
CATALOG INVENT;
```

Catalogs

An SQL/MP catalog is a set of tables and indexes that describe SQL objects. Tables in the set are called catalog tables and SQL creates them—along with their indexes—when you execute a CREATE CATALOG statement.

Each SQL/MP catalog (the set of catalog tables and their indexes) resides on its own Guardian subvolume, and the name of that subvolume is also the name of the catalog. The name has the same form as the subvolume portion of a Guardian file name:

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

For example, \SYS1.\$VOL1.SALES might be the fully qualified name of a catalog with the simple name SALES. If you omit `\node` or `$volume`, SQL uses the current default node and volume to expand the catalog name. (For more information about Guardian names, see [Guardian Names](#) on page G-7.)

Each node on which NonStop SQL/MP is used has one special catalog called the system catalog (described under [System Catalog](#) on page S-92) and might have many other catalogs. Each table, view, index, partition, collation, or catalog table located on a node must be described in a catalog on the same node. Normally, an SQL program is registered in a catalog, too (enabling SQL to locate affected programs when you change definitions of tables, views, indexes, or collations), but you can create unregistered programs if necessary.

A volume can have many catalogs. A subvolume can have only one catalog. Each catalog can describe objects from any subvolume and volume on the same node.

You can create your own tables, indexes, views, and files on a subvolume that includes an SQL catalog, but it is better to create such objects in other subvolumes, if a later release of NonStop SQL/MP could add new tables or indexes to the catalog. (The UPGRADE CATALOG command would be unable to upgrade your catalog if a new catalog table had the same name as a table or file you created on the catalog subvolume.)

Each catalog has a version and a format version associated with it. The catalog version indicates the newest version of objects that can be registered in the catalog. The catalog format version indicates the oldest version of the SQL/MP software that can access the catalog. (For more information, see [Versions](#) on page V-6.)

[Table C-1](#) briefly describes catalog tables and indexes. For more information about a specific catalog table, see the separate entry for that table.

Table C-1. Catalog Tables and Indexes (page 1 of 2)

Table	Function
BASETABS	Describes the attributes of tables
CATALOGS	Describes the catalogs on a node (present only in system catalogs)
COLUMNS	Describes the columns of tables

Table C-1. Catalog Tables and Indexes (page 2 of 2)

Table	Function
COMMENTS	Stores comments on collations, columns, constraints, indexes, tables, and views; also stores help text for columns
CONSTRNT	Describes constraints defined on tables
CPRLSRCE	Stores source for collations
CPRULES	Describes collations
FILES	Describes attributes of files that contain tables and indexes
INDEXES	Describes indexes defined on tables
IXINDE01	Unique index on INDEXNAME column of INDEXES table
IXPART01	Nonunique index on PARTITIONNAME column of PARTNS table
IXPROG01	Nonunique index on GROUPID and USERID columns of PROGRAMS table
IXTABL01	Nonunique index on GROUPID and USERID columns of TABLES table
IXUSAG01	Nonunique index on USINGOBJNAME and USINGOBJTYPE columns of USAGES table
KEYS	Describes the key columns of indexes
PARTNS	Describes partitions of tables and indexes
PROGRAMS	Describes SQL program files
TABLES	Describes tables, views, and collations
TRANSIDS	Stores TMF transaction IDs for current DDL operations on the catalog
USAGES	Describes dependencies among SQL objects
VERSIONS	Keeps version information about the catalog
VIEWS	Describes the attributes of views

Operations on Catalog Tables

You create a catalog with `CREATE CATALOG`. You drop a catalog (after dropping all user-defined collations, indexes, programs, tables, and views that the catalog describes) with `DROP CATALOG`.

You can alter the security of an entire catalog with `ALTER CATALOG`, or alter the security of the `CATALOGS`, `PROGRAMS`, `TRANSIDS`, or `USAGES` catalog tables with `ALTER TABLE`.

You can find the version of a catalog with `GET VERSION` and change the version with `UPGRADE CATALOG` or `DOWNGRADE CATALOG`. (To change the version of the system catalog, use `UPGRADE SYSTEM CATALOG` or `DOWNGRADE SYSTEM CATALOG`.)

SQL automatically updates the contents of catalog tables for you as you execute other statements that create, drop, or modify objects described in the catalog. (A few fields

that contain statistics are updated only when you issue an UPDATE STATISTICS command.)

You can also use these statements with catalog tables:

DML:	DECLARE CURSOR FETCH SELECT
DCL:	CONTROL EXECUTOR CONTROL QUERY CONTROL TABLE (has no effect) FREE RESOURCES LOCK TABLE (not recommended)
DDL:	ALTER CATALOG (security only) ALTER TABLE (security only for CATALOGS, PROGRAMS, TRANSIDS, or USAGES tables) COMMENT CREATE CATALOG CREATE VIEW (shorthand views only) DROP CATALOG UPDATE STATISTICS
Other:	INVOKE DISPLAY USE OF VERIFY

You cannot use other SQL statements on catalog tables. For example, you cannot create indexes or constraints on catalog tables or drop individual tables within the catalog.

△ **Caution.** Normally, DELETE, INSERT, and UPDATE statements do not work on catalog tables, but a licensed SQLCI2 process (or a licensed SQL program file) can use any DML operation on catalog tables, as described in the *SQL/MP Installation and Management Guide*.

Only the most extreme situations should require the use of a licensed SQLCI2 process, because such operations can be extremely dangerous to the consistency of the database and the data dictionary.

CATALOGS Table

The CATALOGS table is a catalog table that describes all the catalogs on a node. CATALOGS is part of the system catalog for the node and does not exist in user-created catalogs.

The CATALOGS table is always located on a subvolume named SQL, even if the system catalog is installed on a subvolume of a different name. As a result, it is sometimes called the SQL.CATALOGS table. It is always on the same volume as the system catalog for the node.

Table C-2. The SQL.CATALOGS Table

Column Name	Data Type	Description
1 CATALOGNAME *	CHAR(25)	Node, volume, and subvolume of a catalog
2 SUBSYSTEMNAME *	CHAR(30)	Name of subsystem where catalog resides
3 VERSION	CHAR(4)	Version of catalog in character format: A010 = Version 1 A011 = Version 2 A3nn = Version 3nn
4 VERSIONUPGRADETIME	LARGEINT SIGNED	Julian timestamp from last upgrade or downgrade of catalog
5 CATALOGCLASS	CHAR(1)	S if system catalog U if user catalog
6 CATALOGVERSION	SMALLINT UNSIGNED	Version number of catalog

* Indicates primary key

The columns CATALOGNAME through CATALOGCLASS (1 through 5) were created in version 1 of NonStop SQL/MP. The sixth column, CATALOGVERSION, was added in version 300.

If you have the required authority, you can use the CATALOGS table to obtain information such as the names of all objects on a node that belong to a particular user. (Retrieve the names of all catalogs from the table, search each catalog for objects owned by the user.)

Guardian names in CATALOGS are fully qualified and use uppercase characters.

The ALTER CATALOG command does not affect the system catalog CATALOGS table. You must alter that table with the ALTER TABLE command.

CENTER_REPORT Option

CENTER_REPORT is an option of the SQLCI report writer SET LAYOUT command that controls whether reports are centered within the left and right margins.

<pre>CENTER_REPORT { OFF } { ON }</pre>
--

Consideration—CENTER_REPORT

If you specify ON, the report writer centers the whole report as a block of text. To compute the report width, the report writer examines lines of the report (including titles, footings, detail lines, and so forth) and determines the leftmost and rightmost positions containing data or text.

During these calculations, the report writer ignores lines defined with a CENTER clause and leading and trailing TAB clauses. (In a TAB clause, the number you specify relates to an absolute character position before the report is centered.)

The default is OFF.

Example—CENTER REPORT

This command centers all reports until you reset the CENTER_REPORT option:

```
>> SET LAYOUT CENTER_REPORT ON;
```


Character Data Types

SQL includes both fixed-length character data and varying-length character data. Either type of character data can be associated with a character set. Any character data type is compatible with any other character data type that is associated with the same character set, but not with numeric, date-time, or interval data types, and not with character data that is associated with a different character set.

Fixed-Length Types	Varying-Length Types
CHAR[ACTER]	CHAR[ACTER] VARYING
PIC X [DISPLAY]	VARCHAR[ACTER]
NATIONAL CHAR[ACTER]	NATIONAL CHAR[ACTER] VARYING
NCHAR[ACTER]	NCHAR[ACTER] VARYING

The maximum length of a character column depends on whether the data type is fixed-length or variable-length, whether the associated character set is single-byte or double-byte, and on the file organization of the file that contains the column:

Data Type	Key- Sequenced	Relative or Entry- Sequenced
Single-byte unvarying	4061	4072
Single-byte VARYING	4059	4070
Double-byte unvarying	2030	2036
Double-byte VARYING	2029	2035

Each variable-length character data item requires two bytes of storage for length information, in addition to the space required for the data itself. As a result, the maximum length for a variable-length column is less than the maximum length for an otherwise equivalent fixed-length column.

A column that allows null values requires two extra bytes.

A string literal can be as long as a character column.

Note that C string data types require an additional trailing null character. Because of this additional null character, using C string data types disables the SQL bulk move feature when you move contiguous data fields between the Executor Extended Data Segment and program host variables.

Character Expressions

A character expression specifies a value and can be a simple string literal or a column name that specifies the value of a column in a row of a table. The expression can include string operators and function calls that return string results. All these are character expressions:

"ABILENE"	A character string
CUSTNAME	The value in column CUSTNAME
SUBSTRING ("Robert" FROM 0 FOR 3)	The SUBSTRING function applied to the string "Robert"

A character expression has a CHAR or VARCHAR data type and can be upshifted.

A character expression consists of one or more operands connected by string operators, as shown in the diagram.

```
string-operand [ [ string-operator
                  string-operand ] ... ]
```

string-operand is:

```
{ string-literal
{ column-name
{ parameter-name
{ host-variable-name
{ string-function-invocation }
```

string-operator is:

```
{ concatenation-operator }
```

string-literal

represents a series of characters and consists of that series of characters surrounded by double or single quotation marks, optionally preceded by a clause that specifies the character set associated with the characters. For information, see [String Literals](#) on page S-80.

column-name

is the valid name of a column in a table, optionally qualified by a collation name. The column name must refer to a column with a character data type.

parameter-name

is a parameter of a character data type, optionally associated with a collation.

host-variable-name

is the name of a host variable that contains a value with a character data type.

string-function-invocation

is a call to a function such as UPSHIFT, TRIM, or SUBSTRING that returns a string as a result.

concatenation-operator

specifies this operator:

||

The concatenation operator concatenates two string operands and produces a string as a result. If either of the character strings has a VARCHAR data type, the result has a VARCHAR data type. If both character strings are fixed CHAR strings, the result is a fixed CHAR. Operands should have identical character sets.

Considerations—Character Expressions

- Guidelines for using character expressions in SQL statements

Character expressions can appear at any place where a string literal, parameter, column, or host variable can appear. This diagram lists the usage of character expressions:

```
{ character-expression relat-operator character-expression }
{ character-expression [NOT] LIKE character-expression }
{ character-expression [NOT] BETWEEN }
{ character-expression AND character-expression }
{ character-expression IS [NOT] NULL }
{ character-expression [NOT] IN { subquery } }
{                               { in-value-list } }
{ character-expression relat-operator [ALL ] subquery }
{                                     [ANY ] }
{                                     [SOME ] }
{ UPSHIFT ( character-expression ) }
```

When evaluating predicates with character expressions, the rules that apply to string literals, columns, and other forms of character expressions also apply to this usage.

- Guidelines for using the concatenation operator

The sum of lengths of the character string operands cannot exceed the maximum allowed length for their data type. If their length exceeds the maximum allowed for a character data type, SQL truncates the string to the right and issues a warning.

Strings with different character sets cannot be concatenated together.

If either of the two character string operands is a null value, the result is a null value.

If a concatenated expression is used in a comparison (as in a WHERE clause), the collation of the resulting expression must not be undefined. If the concatenation is not used in a comparison (as in a select list that is not ordered), it can have an undefined collation.

- Associating collations with character data

To associate a collation with a parameter of a character data type, use the COLLATE command:

```
character-item [ COLLATE { collation } ]
                     { CHARACTER SET }
```

A character expression is implicitly associated with a collation if *character-item* is defined with a collation. For example, if *character-item* is a column that was defined with a COLLATE FRENCH clause, the collation FRENCH is implicitly associated with the expression, although no COLLATE clause appears in the expression.

A character expression is explicitly associated with a collation if the expression itself includes the COLLATE clause. An explicit association with a collation overrides an implicit association with a collation in the same expression.

COLLATE CHARACTER SET explicitly associates the binary ordering of *character-item* values with the expression, overriding any implicit association with a collation.

If a nested character expression includes more than one COLLATE clause, the collation explicitly specified at the highest level of the expression is the collation associated with the expression. For example, this expression is associated with the collation SPANISH, even if B is a column associated with collation FRENCH:

```
MAX(B COLLATE FRENCH) COLLATE SPANISH
```

Collation FRENCH is used to compute the MAX value. The result has collation SPANISH.

- Determining the collating sequence for concatenated strings

The collating sequence of a concatenated string is determined by the rules specified for a comparison operation. For more information, see [Collations](#) on page C-43.

Examples—Character Expressions

- This example on concatenation results in “Robert Smith”:

```
"Robert " || "Smith"
```

Note blanks between names are included in the original string literals.

- This example results in “Robert John Smith”:

```
"Robert " || "John " || "Smith"
```

- This example concatenates “Robert” with a string of length 0, which results in “Robert”:

```
"Robert" || ""
```

- This example results in “Robert SMITH”:

```
"Robert " || UPSHIFT ("Smith")
```

- This example results in “Robert Smith” with the collating sequence FRENCH:

```
"Robert " COLLATE FRENCH || "Smith"
```

- This SELECT statement returns a null value because one of the two character string operands has a null value:

```
>>CREATE TABLE EMPNAME (first_name char(10),
+> last_name char(10));
>>INSERT INTO EMPNAME VALUES ("Robert ", NULL);
>>SELECT ( first_name || last_name ) FROM EMPNAME;
```

Character Sets

NonStop SQL/MP allows you to associate one of these character sets with a column, a literal, a host variable, or a parameter:

- ISO 8859/1 through ISO 8859/9
- Kanji
- KSC5601

You can also define a collation that uses any of the nine ISO 8859 character sets and associate the collation with a column, a literal, an host variable, or a parameter of the same character set. (You cannot define a collation that uses the Kanji or KSC5601 characters sets. SQL always collates characters from those character sets according to the binary value of the characters.)

For compatibility with versions of NonStop SQL/MP that do not support multiple character sets, you can specify UNKNOWN to indicate that the character set is unknown. SQL considers this equivalent to omitting the character set specification and handles the data as 8-bit data.

ISO 8859 Character Sets

The ISO 8859 character sets are a standard set of nine single-byte character sets defined by ISO (the International Organization for Standardization) in a series called ISO 8859. The first in the series is called ISO 8859/1, the second is ISO 8859/2, and so on through ISO 8859/9. In NonStop SQL/MP, you use the keywords ISO88591, ISO88592, ISO88593, and so forth to specify a character set within the ISO 8859 series.

ISO 8859 defines printing characters for each character set, and all character sets share the same layout. Each set includes graphic characters from the ASCII character set (a 7-bit character set defined in both ISO and ANSI standards) in code positions %H20-%H7E and other characters in positions %HA0-%HFF, allowing 96 graphic characters to be added to those already in ASCII. Graphic characters that appear in multiple ISO 8859 character sets always have the same encoding.

The ranges %H00-%H1F and %H7F-%H9F are reserved for control characters, but ISO 8859 does not make specific control character assignments.

ISO 8859/1, which is informally called Latin-1, is the most commonly used ISO 8859 character set. ISO 8859/1 contains the characters necessary for Western European languages such as French, German, Italian, and Spanish. It is HP's current default character set and is implemented in the most recent version of 6525A terminal, PCT, and printers.

The other ISO 8859 character sets are used in varying degrees throughout the world. ISO 8859/2 is used for Eastern European languages, ISO 8859/3 for Southeastern European languages, ISO 8859/4 for Northern European languages, ISO 8859/5 for English and Cyrillic languages, ISO 8859/6 for English and Arabic languages, ISO 8859/7 for English and Greek languages, ISO 8859/8 for English and Hebrew languages, and ISO 8859/9 for Western European and Turkish languages.

The nine ISO 8859 character sets are not completely documented in SQL/MP documentation. However, for information on the ASCII characters common to all nine ISO 8859 sets, see [ASCII Character Set](#) on page A-70.

Kanji Character Set

The Kanji character set (also known as the Shift JIS character set or—in NonStop system use—the HP Kanji character set) is a double-byte character set originally developed for CP/M microcomputers in Japan and later adopted for use on MS-DOS systems. The Kanji character set is in common use on a variety of Japanese mainframes as well. In NonStop SQL/MP, you use the keyword KANJI to specify the Kanji character set.

NonStop SQL/MP always collates Kanji characters according to their binary values.

Neither byte of any Kanji character contains any binary value less than %H40, as shown:

First-byte range	%H81 .. %H9F, %HE0 .. %HFC
Second-byte range	%H40 .. %H7E, %H80 .. %HFC
Number of possible characters	11280
Two-byte space character	%H8140
One-byte space character	%H20

KSC5601 Character Set

The KSC5601 character set (also known as the Korean character set or—in NonStop system use--the HP KSC5601 character set) is the double-byte character set that is the Korean Industrial Standard character set and is required on systems used by government and banking sectors within Korea. In NonStop SQL/MP, you use the keyword KSC5601 to specify the KSC5601 character set.

NonStop SQL/MP always collates KSC5601 characters according to their binary values.

Neither byte of any KSC5601 character contains any binary value less than %H40, as shown:

First-byte range	%HA1 .. %HFE
Second-byte range	%HA1 .. %HFE
Number of possible characters	8836
Two-byte space character	%HA1A1
One-byte space character	%H20

CHAR_LENGTH Function

The CHAR_LENGTH function returns the number of characters in a string.

```
CHAR[ACTER]_LENGTH ( character-string )
```

where *character-string* is:

```
{ string-literal           }
{ column-name             }
{ param-name              }
{ host-var-name           }
{ UPSHIFT function         }
{ character-expression    }
```

character-string

specifies the string for which the length is to be returned.

Considerations—CHAR_LENGTH Function

- SQL returns the result as a two-byte signed integer with a scale of zero.
- If *character-string* is a null value, SQL returns a length of null.
- For a column declared as a fixed CHAR, SQL returns the maximum length of that column. For a VARCHAR column, SQL returns the actual length of the string stored in that column.
- The OCTET_LENGTH and CHAR_LENGTH functions are similar. The OCTET_LENGTH function returns the number of bytes in the string. The result of both functions is the same for single-byte character data types. For a multi byte character data type, the two functions return different results.

Examples—CHAR_LENGTH Function

- This example returns 12:

```
CHAR_LENGTH ( "Robert" || " " || "Smith" )
```

- This example returns zero:

```
CHARACTER_LENGTH ( " " )
```

- This example returns the value 3:

```
CHAR_LENGTH ( _KANJI "abcdef" )
```

- These two examples use a table created:

```
CREATE TABLE EMPLOYEE (EMPNAME CHAR(20),
                        ADDRESS VARCHAR(100));
INSERT INTO EMPLOYEE VALUES ("Robert Smith",
                              "19333 Vallco Parkway ");
```


- This example returns 20:

```
CHAR_LENGTH ( EMPNAME )
```

- This example returns 21—not 100—because it is a VARCHAR value:

```
CHAR_LENGTH ( ADDRESS )
```

CLEANUP Command

CLEANUP is an SQLCI utility command that allows a user with super ID authority to delete damaged SQL objects, SQL programs in Guardian files, and catalogs and associated file labels and shadow labels from the local node.

△ **Caution.** Use CLEANUP only when absolutely necessary to delete damaged objects. CLEANUP deletes both damaged and undamaged objects in the fileset you specify. Misusing CLEANUP can corrupt your SQL data dictionary. Never use CLEANUP as a substitute for DROP or PURGE.

```
CLEANUP [!] qualified-fileset-list [!] [ , option ] ;
```

```
option is: { CATALOG[S]
             { NO CATALOG[S] [ , SHADOWONLY ] }
             { SHADOWONLY [ , NO CATALOG[S] ] }
```

qualified-fileset-list

is a qualified fileset list that specifies SQL objects to delete. (For information, see [Qualified Fileset List](#) on page Q-1.) If SMF is installed, *qualified-fileset-list* cannot specify a file on a \$*.ZYS*. subvolume.

CLEANUP does not delete catalog tables and indexes included in *qualified-fileset-list* unless you also specify the CATALOG[S] option. CLEANUP does not delete Enscribe files other than SQL programs. CLEANUP does not delete OSS files, even if they contain SQL programs.

If an object's file label no longer exists, you can use the FROM CATALOG clause to specify the object, but you cannot use a WHERE expression. If you use both, CLEANUP issues an error message and does not delete the object.

!

(either before or after *qualified-fileset-list*) directs CLEANUP to delete all objects in the specified filesets without prompting for confirmation.

If you omit the exclamation point when you use CLEANUP interactively, this prompts normally appear:

```
DO YOU WISH TO CLEANUP THE ENTIRE FILESET
    ... (name of fileset) ...
(Y[ES], N[ONE], S[ELECT], F[ILES])?
```

If you allow CLEANUP to prompt, use one of these responses:

Y[ES]	Purge the whole set
N[ONE]	Cancel the command
S[ELECT]	Prompt object-by-object
F[ILES]	List objects in fileset

[NO] CATALOG[S]

specifies whether to delete catalog tables and associated indexes in addition to other objects and programs specified in *qualified-fileset-list*.

To purge catalog tables and indexes, you must specify CATALOG[S] and you must use the wild-card character (*) in the *qualified-fileset-list* to indicate all catalog tables and indexes in the catalog. (CLEANUP can purge only entire catalogs, not individual catalog tables or indexes within a catalog.)

If you omit CATALOG[S] or specify NO CATALOG[S], CLEANUP deletes only objects that are not part of a catalog.

SHADOWSONLY

directs SQL to delete shadow labels for objects in *qualified-fileset-list*, but not the objects themselves. If you omit SHADOWSONLY, CLEANUP does not purge shadow labels. (Shadow labels—temporary internal labels created when objects are dropped from the node—are discussed in the *SQL/MP Installation and Management Guide*.)

You cannot specify SHADOWSONLY and CATALOGS in the same CLEANUP command.

Considerations—CLEANUP

- Only the local super ID can use CLEANUP. To delete damaged objects on multiple nodes, the super ID on each node must run CLEANUP for that node.
- After deleting each object, CLEANUP displays a message indicating the object and catalog entry that was deleted.
- You cannot use CLEANUP within a user-defined TMF transaction.

SQL automatically starts a transaction for each catalog description and file label purged with CLEANUP, so only the deletion of the last SQL object (or partition) is undone if CLEANUP fails before the deletion is committed.

Catalog description and file label purging are two different and independent activities. CLEANUP is designed to handle these variations:

- Purging both the file label and the corresponding catalog entry, if they exist, for the object program file
- Purging only the catalog entry for the object or program file
- Purging only the file label for the object or program file

If CLEANUP is interrupted by a break request, all changes that have completed at the time of the break remain in effect; any change in progress is rolled back. The messages displayed by CLEANUP before the break is received indicate which objects have been fully deleted. In the rare case when the break arrives immediately after a deletion and just before a message is issued, CLEANUP actually might have deleted one additional object.

- SQL objects are described in the SQL data dictionary, which is composed of SQL catalogs and file labels. Misuse of various system management utilities can corrupt the data dictionary. This damage can make it impossible to access the objects, preventing you from removing them with the customary DROP command or PURGE utility. In such cases, you can probably eliminate the objects by using CLEANUP.

When purging an object, CLEANUP attempts to purge the file containing the object and the description of the object in the catalog.

For a table, index, or view specified in the fileset list, CLEANUP attempts to purge dependent objects and to mark dependent programs as invalid. This operation is consistent with those of the DROP and PURGE commands; however, unlike DROP and PURGE, CLEANUP deletes each object independently of the other objects.

For an SQL program in a Guardian file specified in the fileset list, CLEANUP purges the program. CLEANUP cannot operate on an SQL program in an OSS file.

For partitioned objects, CLEANUP processes each partition as a separate object, purging each one independently of the other partitions. You cannot request a CLEANUP operation for a single partition.

CLEANUP also processes each dependent object independently. Under unusual circumstances it is possible to still have pieces of the dependent objects, partitions, or indexes remaining (after using CLEANUP) that refer to a deleted table.

For indexes, the CLEANUP utility does not update the object version of any dependent object of the purged index. The recorded object versions of some objects, therefore, might not be the same as the actual object versions.

For collations, CLEANUP does not purge a collation or its description if the collation has dependent objects. If *qualified-fileset-list* includes both a collation and all its dependent objects, SQL purges the dependent objects first, the collation and its description. If *qualified-fileset-list* includes an object

that depends on a collation but not the collation, CLEANUP purges the dependent object and deletes the relationship to the collation.

- CLEANUP cleans up objects only on the local node. To delete all partitions of a damaged object distributed over several nodes, you must run CLEANUP on each of the nodes involved.

△ **Caution.** Never use this command to purge SQL objects and catalogs:

```
SQLCI CLEANUP $*.*.*!, CATALOGS;
```

This command deletes all SQL objects and catalogs from a node, including the system catalog and the \$SYSTEM.SYSTEM.SQLCI2 program. If you want to remove NonStop SQL/MP from a node, see the *SQL/MP Installation and Management Guide* for instructions.

CLEANUP Exception Cases

In certain situations (described in this text) the catalog does not contain enough information to enable CLEANUP to remove the catalog description of an SQL object. In such cases, you can use a licensed SQLCI2 process to manually correct the catalog and rerun CLEANUP.

If CLEANUP is unable to remove an object's file label, you can use the GOAWAY utility to remove the file label, but you should use GOAWAY only as a last resort.

Specifically, you cannot use CLEANUP to:

- Purge a view when the entry for the view in the TABLES table cannot be accessed.
- Purge a program when the entry for the program in the PROGRAMS table cannot be accessed.
- Purge an object that has no file label and no entry in the USAGES table.
- Purge an object when the entry for the object's catalog in the VERSIONS table cannot be accessed.
- Purge a dependent object that you do not explicitly name in *qualified-fileset-list* if USAGES table information about the object is missing or inaccessible.
- Purge a catalog's name entry in the system catalog's CATALOGS table when the BASETABS and COLUMNS tables for the catalog are missing.

When purging a catalog, CLEANUP does not remove the catalog's name entry from the system catalog CATALOGS table until the catalog's BASETABS or COLUMNS table is purged. If, however, the BASETABS and COLUMNS tables are already missing when you request the CLEANUP operation, CLEANUP removes the remaining portion of the catalog but does not delete the name entry in the system catalog's CATALOG table. To reuse the purged catalog's name, you must first remove the orphan entry from the system CATALOGS table with a licensed SQLCI2 process.

- Purge a catalog's name entry in the system catalog's CATALOGS table when a file-system error occurs during access to the system CATALOGS table. When CLEANUP is used to purge a catalog, CLEANUP attempts to remove the name entry for the catalog from the system CATALOGS table. Any failure in this operation (caused by an event such as a system catalog file being flagged as CRASHOPEN) does not affect the outcome of the current CLEANUP operation. CLEANUP reports a warning to the user if the catalog's name entry cannot be removed because of a file-system error. In this case, you must use a licensed SQLCI2 process to remove the catalog name entry from the system CATALOGS table.

- Purge the catalog information and the file label of any protection view whose underlying table or table partition is missing or inconsistent.

Normally, a protection view is purged when the underlying table is removed by CLEANUP; however, misuse of the GOAWAY utility on a base table or another software problem could cause an orphan protection view. In this situation, if the entire catalog is not being removed, you must use a licensed SQLCI2 process to remove the catalog description associated with the protection view and use GOAWAY to remove the file label of the view.

- Purge a dependent partitioned protection view when the catalog tables describing the view are missing. This situation occurs when a table with a protection view is partitioned across multiple catalogs, and one of the catalogs in which a secondary partition is registered is missing when CLEANUP is requested to purge the partitioned table. In this case, CLEANUP cannot determine the dependents of the table partition whose catalog is missing because the USAGES table is absent; this situation can cause an orphaned protection view. To remove the orphaned protection view, you must use the GOAWAY utility.
- Purge catalog entries for SQL objects or determine dependent objects when the VERSIONS table is missing or corrupted. The catalog tables themselves, however, can still be purged in this case.
- Purge catalog information, in certain unusual circumstances, for a partition of a partitioned index when the file label of the partition is missing. CLEANUP, however, successfully removes the file labels and catalog descriptions of all other partitions of the index.

Example—CLEANUP

Suppose that the subvolume \$VOL1.PERSNL contains three tables: DEPT, JOB, and EMPLOYEE. The tables are described in the catalog \$VOL2.CAT. If \$VOL2 is removed, the tables cannot be deleted using DROP or PURGE because the catalog in which they are described is not accessible. You can, however, remove the tables by entering:

```
>> CLEANUP ($VOL1.PERSNL.DEPT, $VOL1.PERSNL.JOB,
           $VOL1.PERSNL.EMPLOYEE);
```

If \$VOL1 is removed (instead of \$VOL2, as in the previous example), the catalog descriptions cannot be removed using DROP or PURGE because the tables are not accessible. You can delete the catalog descriptions by entering:

```
>> CLEANUP ($VOL1.PERSNL.DEPT, $VOL1.PERSNL.JOB,
           $VOL1.PERSNL.EMPLOYEE) FROM CATALOG $VOL2.CAT;
```

If the DEPT, JOB, and EMPLOYEE tables are the only SQL objects on \$VOL1, you can accomplish the same operation:

```
>> CLEANUP $VOL1.PERSNL.* FROM CATALOG $VOL2.CAT;
```

CLEARONPURGE File Attribute

CLEARONPURGE is a Guardian file attribute that controls erasure of data from the disk when a table, index, catalog, or program is purged or dropped. CLEARONPURGE applies to key-sequenced, relative, and entry-sequenced tables and to indexes.

{ CLEARONPURGE NO CLEARONPURGE }

NO CLEARONPURGE is the default for tables, for catalogs, and for programs that are explicitly SQL-compiled and stored in Guardian files.

The index default is the table value at index creation.

Considerations—CLEARONPURGE

- When you drop or purge an object with NO CLEARONPURGE, the system deallocates disk space but does not physically destroy the data in that disk space. This implementation improves performance by reducing writes to the disk, but when the disk space is allocated to a new file, other users might be able to read data left by the object that used the space previously.

CLEARONPURGE increases security for sensitive data or programs by causing the system to overwrite deallocated disk space.

- If you drop or purge a file with the CLEARONPURGE attribute from within a TMF transaction, the data is not physically erased from the disk until after the transaction commits.

CLOSE Statement

CLOSE is a DML and a dynamic SQL statement that closes a cursor in a host program. After the CLOSE executes, the result table for the cursor (the output that results from the execution of the SELECT for the cursor) no longer exists.

CLOSE { <i>cursor</i> : <i>cursor-variable</i> }

cursor

is the name of an open cursor to close.

:cursor-variable

is a host variable of type CHAR or VARCHAR that stores the name of an open cursor to close.

Considerations—CLOSE

- There are no authorization requirements for closing a cursor.
- Closing a cursor defined with REPEATABLE access does not affect locks. Locks on audited tables are released when the TMF transaction finishes or aborts; locks on nonaudited tables must be released with UNLOCK TABLE.

Closing a cursor defined with STABLE access for an audited table releases the row lock acquired on the last FETCH only for a row that was not updated or deleted using the cursor; locks on rows that were updated or deleted are not released until the TMF transaction ends.

Closing a cursor defined with STABLE access for a nonaudited table releases the row lock acquired on the last FETCH.

- COMMIT WORK automatically closes cursors that reference audited tables. You can use CLOSE only on a cursor for an audited table using STABLE or REPEATABLE access within the same TMF transaction as the last FETCH on the cursor. (If no FETCH statements execute after the cursor is opened, you can use CLOSE on the cursor in any transaction or outside of a transaction.)
- If your program is a server and the TMF transaction was started in a requester, the program must close cursors to release space used by the cursors and to free locks before returning control to the requester.
- You can use the FREE RESOURCES statement instead of the CLOSE statement to close all open cursors. After the cursor is closed, you must reopen the cursor before referring to it in any statement.

Example—CLOSE

This program fragment declares and opens a cursor, uses FETCH to retrieve data, closes the cursor:

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR
      SELECT COL1, COL2, COL3 FROM =PARTS
      WHERE COL1 >= :HOSTVAR1 ORDER BY COL1 BROWSE ACCESS;
EXEC SQL OPEN CURSOR1;
EXEC SQL FETCH CURSOR1 INTO :HOSTVAR1, :HOSTVAR2, :HOSTVAR3;
EXEC SQL CLOSE CURSOR1;
```

Clustering Keys

A clustering key is the user-defined portion of a primary key that is determined partly by the user and partly by the system. Values for a clustering key do not need to be unique, as required for user-defined primary keys. Only key-sequenced tables can have clustering keys.

To define a clustering key, specify one or more columns in the CLUSTERING KEY clause of the CREATE TABLE statement. SQL adds a column named SYSKEY (data type LARGEINT SIGNED and ASCENDING sort order) as the first column of the table and uses a primary key that consists of the clustering key you specified concatenated with the SYSKEY. (SYSKEY is the first column of the table but the last column in the primary key.)

When you add a row to the table, the file system automatically generates a unique 8-byte number as a value for the SYSKEY column, enabling SQL to uniquely identify the row. You cannot specify the value for the SYSKEY column; it is always supplied by the file system.

The primary key for a table with a clustering key is the column or columns in the user-defined clustering key followed by the system-defined SYSKEY column. With SYSKEY added to the clustering key, the primary key has a unique value for each row. Because the SYSKEY value cannot be specified by an application, do not use a clustering key if you need a unique key that can be supplied by your application.

Like other columns in a primary key, columns in a clustering key cannot be updated and cannot contain null values. The combined length of the columns in a clustering key, not including the 8-byte SYSKEY column, cannot exceed 247 bytes.

The catalog description of a table with a clustering key reflects the presence of the 8-byte SYSKEY column, but SQL does not display SYSKEY as part of the table unless a query explicitly selects the SYSKEY column. In a table that includes a SYSKEY column, for example, this SELECT statement does not display SYSKEY:

```
SELECT * FROM table-name
```

In a view definition, however, the same SELECT includes SYSKEY in the view columns unless you are using a protection view. For more information, see [Protection View](#) on page P-32.

Example—CLUSTERING KEYS

This statement declares a table with a clustering key:

```
CREATE TABLE CK (SYS_ID SMALLINT, CPU SMALLINT, PIN SMALLINT,  
  PROG_NAME VARCHAR(34)) CLUSTERING KEY (SYS_ID, CPU, PIN);
```


COLLATE Clause

The COLLATE clause associates an existing collation with a character expression, with a column of a character data type that is being added to a table, or with an index that is being created.

For information about using the COLLATE clause on a character expression, see [Character Expressions](#) on page C-14.

For information about using the COLLATE clause with a data type specification for a CREATE TABLE or ALTER TABLE statement, see [Data Types](#) on page D-1.

For information about using the COLLATE clause to associate a collation with an index, see [CREATE INDEX Statement](#) on page C-142.

For general information about collations, see [Collation Definitions](#) on page C-30.

For information about creating collations, see [CREATE COLLATION Statement](#) on page C-137 and [Collation Definitions](#) on page C-30.

Collation Definitions

A collation definition is a description of a collating sequence that can be written in an EDIT file and processed by the CREATE COLLATION statement to create an SQL collation.

The simplest possible collation definition consists of an LC_COLLATE section that includes an ordered list of elements in the collation.

More complex collation definitions can also include comments, redefine the comment and escape characters, define multicharacter collation elements in the LC_COLLATE section, define character classes and upshifting rules in the LC_CTYPE section, and specify a character set for the collation in the LC_TDMCODESET section.

The language in which you express a collation definition is based on the POSIX/XPG4 standard, so you can take a localedef source file from an X/Open Locale Registry and create an SQL/MP collation definition with only minimal modifications. The language follows completely different syntactic and semantic rules from SQL statements or SQLCI commands. One major difference is that case is significant in keywords within collation definitions.

The remainder of this entry describes the collation definition language, beginning with rules for comment and escape characters, followed by rules for each of the three major sections within a collation description (the LC_COLLATE section, the LC_CTYPE section, and the LC_TDMCODESET section), and ending with examples and special considerations.

Remember that keywords shown in uppercase must be entered in uppercase, and keywords shown in lowercase must be entered in lowercase. Also note that angle brackets appear in several parts of the collation definition language as an element of the language itself to represent variable items you must supply, as they are commonly used elsewhere in this documentation.

Comment and Escape Characters in Collation Definitions

The default comment character is the number sign (#).

The default escape character is the backslash (\).

Use the comment character to include comments in a collation definition. All characters between the comment character and the end of a physical line are handled as a comment, including the escape character.

You use the escape character to continue a clause over more than one physical line. If you specify the escape character as the last character in a physical line, this line is handled as a continuation of the line that ended with the escape character. You also use the escape character to indicate the beginning of an octal, decimal, or hexadecimal code that represents a character within the collation, as explained under [The LC_COLLATE Section of a Collation Definition](#) on page C-32.

To change the comment character or escape character, specify a new character in angle brackets in a `comment_char` or `escape_char` clause at the beginning of the collation definition. For example, this statement changes the comment character to \$ and the escape character to @:

```
comment_char <$>
```

```
escape_char <@>
```

The `comment_char` and `escape_char` clauses can appear in any order, but each clause, if used, must appear before the main sections of the collation definition. Each can appear only once in the entire collation definition and must be on its own physical line.

You cannot specify the backslash (\) or the escape character as the comment character. You cannot specify the number sign (#) or the comment character as the escape character.

The escape character, the comment character, and these punctuation characters have special meanings in the collation definition language:

,	Comma
;	Semi-colon
"	Quotation mark
.	Period
(Left parenthesis
)	Right parenthesis
<	Left angle bracket
>	Right angle bracket

To specify one of these characters as a character in a collation (rather than as a syntactic element in the collation definition language), precede the character with the escape character.

If you redefine the comment or escape character as one of the punctuation characters just listed, an error occurs if you later use portions of the collation definition language that include these characters as syntactic elements. As a result, you should normally select characters other than those listed as the comment character or escape character.

The LC_COLLATE Section of a Collation Definition

The LC_COLLATE section defines multicharacter elements of the collation and specifies the order of character and multicharacter elements within the collation. The LC_COLLATE section is the only required section in a collation definition.

The LC_COLLATE section can appear only once in a collation definition. Each of the six types of lines that make up the section must begin on a new physical line. Most types of lines can appear only once in the section. If a specific type of line can appear more than once, that is noted in its description.

```
LC_COLLATE

[ collating-element <new-element> from "char char" ]

order_start [ forward ]

element [ weight ]

order_end

END LC_COLLATE
```

LC_COLLATE

starts the LC_COLLATE section.

collating-element <new-element> from "char char"

defines a new multicharacter element for the collation. The angle brackets are a required part of the syntax for *new-element*, but not for the remaining portion of the clause. For example, either of these clauses defines a new collating element <CH> from the combination of the characters *C* and *H*:

```
collating-element <CH> from "CH"
collating-element <CH> from "\d67\d72"
```

The first *char* in the pair cannot be the space character. In addition, if the first *char* in the pair is in octal, hexadecimal, or decimal format and the second *char* represents a digit in the corresponding base (octal, hexadecimal, or decimal), you cannot use simple format for the second *char*. (For information of each format, see the description of *char* on page [C-33](#).)

You can specify multiple collating-element clauses (subject to the general limits described under Considerations sub-section on page [C-38](#)), but each one must appear on a separate line and all such clauses must precede the *order_start* clause.

<new-element>

is a stream of 2 to 30 characters, enclosed in a set of angle brackets, that represents the new element.

char

is a symbol for a character, from one of the single-byte character sets supported by NonStop SQL/MP, in one of these forms:

Form	Example*
Simple (any single printable character)	A
Bracketed (a single character in angle brackets)	<A>
Octal (an escape character followed by two or more octal digits)	\101
Hexadecimal (an escape character followed by “x” and two or more hexadecimal digits)	\x41
Decimal (an escape character followed by “d” and two or more decimal digits)	\d65

* Each example specifies uppercase A from the ASCII character set (a subset of the nine character sets ISO88591 through ISO88599).

You can use any form shown to define a collation element to NonStop SQL/MP, but the bracketed form is more portable because it does not depend on a specific character set, as do the octal, hexadecimal, and decimal forms.

order_start [*forward*]

starts the ordered list of elements in the collation. The ordered list defines the elements of the collating sequence in ascending order.

“forward” is an optional keyword that specifies that comparison operations for the weight level proceed from the start to the end of the string, which is always true for SQL/MP collations. (SQL/MP collation definitions allow you to include the “forward” option for compatibility with the POSIX/XPG4 standard.)

element [*weight*]

specifies an element in the collating sequence and, optionally, a relative position in the collating sequence to use as a weight for the element.

You can specify multiple *element* [*weight*] clauses, subject only to the limits defined in the Considerations sub-section of this entry.

element

is a *char*, a previously defined *new-element*, an ellipsis (...), or the keyword UNDEFINED.

If *element* is an ellipsis, it specifies the ordered series of characters in the character set between the preceding *element* specified in the collation definition

and this *element* in the collation definition. Neither the preceding *element* nor this *element* can be an ellipsis.

If *element* is UNDEFINED, it specifies all characters in the character set not previously included in the ordered list (either directly or with an ellipsis). You can use UNDEFINED only as the last *element* before *order_end*. The only weight allowed with UNDEFINED is IGNORE.

weight

is a *char* or *element* that appears earlier in the ordered list and that specifies a relative position in the collating sequence as a weight for the corresponding *element*. *weight* can also be an ellipsis (...) or the keyword IGNORE.

If you omit *weight* for a collation element, that element represents itself in the collation sequence.

If *weight* is an ellipsis, *element* must also be an ellipsis and each character specified by the ellipsis in the element column has the unique weight for that character.

If *weight* is IGNORE, SQL handles the corresponding *element* as if it does not exist during a comparison between two strings. For example, specifying this three *element [weight]* pairs:

```
<a>
<b>  IGNORE
<c>
```

causes SQL to handle the strings “aacba” and “aaca” as equal in a comparison that uses the collation.

Note that each character in a given character set (ISO88591, for example), has a unique value as a physical weight. The values of the physical and logical weights of a character might differ when a user specifies a weight in a collating sequence.

order_end

ends the ordered list of elements in the collation.

END LC_COLLATE

ends the LC_COLLATE section.

The LC_CTYPE Section of a Collation Definition

The LC_CTYPE section defines character classes and case conversion rules.

The LC_CTYPE section can appear only once in a collation definition. Except for the line that begins with *class-name*, each of the five types of lines shown in the syntax diagram can appear only once in the section. Each type of line must begin on a new physical line.

```
LC_CTYPE

[ charclass class-name [ ; class-name ] ...

    class-name [class-element [ ; class-element ] ... ] ]

[ toupper (lower,upper) [ ; (lower,upper) ] ... ]

END LC_CTYPE
```

LC_CTYPE

starts an LC_CTYPE section.

charclass *class-name* [; *class-name*] ...

specifies names for up to 100 user-defined character classes.

NonStop SQL/MP does not use character classes in any way, but does scan the character class clause for correctness.

Each *class-name* can contain up to 30 letters or digits, but the first character must be a letter. *class-name* must be unique within the collation definition and cannot be any of these words:

charclass	ISO88591	ISO88596	LC_CTYPE
forward	ISO88592	ISO88597	LC_TDMCODESET
END	ISO88593	ISO88599	toupper
from	ISO88594	ISO88599	UNDEFINED
IGNORE	ISO88595	LC_COLLATE	UNKNOWN

class-name [*class-element* [; *class-element*] ...]]

specifies the set of characters from the character set to include in the class *class-name*. *class-element* is normally a *char*, as defined in the discussion of the LC_COLLATE section.

A *class-element* between two *char* symbols in the list can also be an ellipsis, signifying the series of characters between the characters represented by the *char* symbols within the character set.

Specify a *class-name* clause for each *class-name* you list on the charclass clause.

```
toupper ( lower, upper ) [ ; ( lower, upper ) ] ...
```

specifies a set of *char* pairs that defines the relationship between lowercase and uppercase characters.

char is defined in the discussion of the LC_COLLATE section. *lower* specifies a lowercase *char*; *upper* specifies an uppercase *char*.

If neither *char* in a pair has appeared previously in the toupper clause, the first character in the pair is handled as a lowercase character in the collation, and the second character in the pair is handled as the corresponding uppercase character.

If you omit the toupper clause, SQL associates lowercase characters “a” to “z” with uppercase characters “A” to “Z”.

You can define a character as the uppercase version of more than one lowercase letter, or as the lowercase version of more than one uppercase letter. This is useful, for example, if you define a collation for the French language, in which accented lowercase letters often lose their accents if upshifted.

If the *upper* character in a pair appeared in a previous toupper pair, the latter pair defines an upshift only. For example, in this toupper clause (which defines case shifting for the French e-grave character):

```
toupper ( e, E ); \
        ( \d232, E ); \
        ( \d232, \d200 ) # 232 is e-grave; 200 is E-grave
```

the toupper pair (\d232, E) specifies that e-grave upshifts to E but does not specify how to downshift E; the toupper pair (\d232, \d200) specifies that E-grave downshifts to e-grave, but does not specify how to upshift e-grave.

Similarly, if the *lower* character in a toupper pair appears in a previous pair, the latter pair defines a downshift only.

At least one of the characters in each toupper pair must not have appeared in an earlier pair.

```
END LC_CTYPE
```

ends an LC_CTYPE section.

The LC_TDMCODESET Section of a Collation Definition

The LC_TDMCODESET section specifies the character set for the collation. If you do not specify the LC_TDMCODESET clause, SQL uses the ISO88591 character set for the collation.

The LC_TDMCODESET section can appear only once in a collation definition. Each of the three types of lines shown in the syntax diagram can appear only once in the section. Each type of line must begin on a new physical line.

```
LC_TDMCODESET

{ ISO88591 }
{ ISO88592 }
{ ISO88593 }
{ ISO88594 }
{ ISO88595 }
{ ISO88596 }
{ ISO88597 }
{ ISO88598 }
{ ISO88599 }
{ UNKNOWN }

END LC_TDMCODESET
```

LC_TDMCODESET

starts an LC_TDMCODESET section.

ISO88591 ... ISO88599

specifies a single-byte character set supported by NonStop SQL/MP.

UNKNOWN

specifies that the character set is unknown.

END LC_TDMCODESET

ends an LC_TDMCODESET section.

Considerations—Collation Definitions

- Collation limits

A collation can have up to:

- 500 strings
- 100 character classes
- 8192 tokens (keywords, identifiers, punctuation elements, integers, and strings)

- Collations for Pathmaker applications

Collations for Pathmaker-SQL applications must specify the hexadecimal character ff as the last entry in the collation list.

This requirement exists because Pathmaker generates SQL queries in this format:

```
SELECT x FROM t WHERE coll >= :h1
                        AND coll <= :h2
```

where: *h1* is a value entered from a screen, and *:h2* is the same value padded with binary 1s.

Such a query depends on binary 1 having the maximum character value (that is, having the integer value 255 and being positioned last in the order list). An SQL collation supports this usage only if hexadecimal ff is the final character in the order list.

Examples—Collation Definitions

- This example shows an order list from an LC_COLLATE section that includes a German a-umlaut and preserves the relative positions of <a> and <e> in the order of the character collation sequence:

```
LC_COLLATE
order_start  forward
    <a>
    ...
    <e>
    ...
    <z>
    \d196    "<a><e>"
order_end
END LC_COLLATE
```

- This example demonstrates the use of ellipsis and the effect of specifying *weight* for elements in the order list of an LC_COLLATE section.

The ellipsis in the column on the left specifies all the characters between the letters B and Z (that is, C through Y). The ellipsis in the column on the right specifies that all the characters between the letters B and Z collate according to their relative

positions in the collating sequence. The letter b collates the same as itself, although the weight symbol is omitted.

```
LC_COLLATE
order_start    forward
    A          A
    B          B
    ...        ...
    z          Z
    a          A
    b
order_end
END LC_COLLATE
```

- This example also demonstrates the use of ellipsis and *weight* in an order list of an LC_COLLATE section, but uses them differently than the previous example.

The first ellipsis specifies that all the characters between the letters a and z have unique weights equal to the relative order of the characters. The second ellipsis specifies that all the characters between the letters A and E have the same weight as the letter a. The third ellipsis specifies that all the characters between the letters E and N have unique weights equal to the relative order of the characters.

```
LC_COLLATE
order_start    forward
    <a>
    ...
    <z>
    <A>
    ...      <a>
    <E>      <e>
    ...      ...
    <N>      <n>
order_end
END LC_COLLATE
```

- This sample collation definition includes an LC_COLLATE clause, an LC_CTYPE clause, and an LC_TDMCODESET clause:

```

LC_COLLATE
# This case insensitive collating sequence sorts most
# of
# the accented forms of a, e, i, o, and u equal to the
# unaccented form.

# 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.

# The actual collating sequence starts here:
order_start  forward
\d032        \d032    # 32 is the space character
\d160        \d032    # NBSP (non breaking space)
<0>          <0>
...
<9>          <9>
<A>          <A>
...
<Z>          <Z>
<a>          <A>
...
...

<z>          <Z>
\d192        <A>      # 192 - 195 and 224 - 227
...          <A>      # are forms of "A" and "a"
\d195        <A>
\d224        <A>
...          <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 and 232 - 235
...          <E>      # are forms of "E" and "e"
\d203        <E>
\d232        <E>
...          <E>
\d235        <E>
\d204        <I>      # 204 - 207 and 236 - 239
...          <I>      # are forms 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 and 242 - 245
...        <O>      # are forms of "O" and "o"
\d213      <O>
\d242      <O>
...        <O>
\d245      <O>
\d217      <U>      # 217 - 219 and 249 - 251
...        <U>      # are forms of "U" and "u"
\d219      <U>
\d249      <U>
...        <U>
\d251      <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
\d096      <.>
\d123      <{>      # 123 - 126 are symbols
...        ...      # encoded in sequences
\d126      <~>
\d127      IGNORE

\d196      "<a><e>" # A-umlaut sorts as string of a
and e
\d228      "<a><e>" # a-umlaut sorts as string of a
and e
\d214      "<o><e>" # O-umlaut sorts as string of o
and e
\d246      "<o><e>" # o-umlaut sorts as string of o
and e
\d220v     "<u><e>" # U-umlaut sorts as string of u
and e
\d252      "<u><e>" # u-umlaut sorts as string of u
and e

```

```

    \d223      "<s><s>" # sharp-s sorts as string of s
and s
    \d163      \d163      # upper-half specials and
controls
    \d215      \d215      # multiply sign
    \d159      \d159
    \d170      \d170
    \d186      \d186
    UNDEFINED  IGNORE
order_end
END LC_COLLATE

LC_CTYPE
charclass  alphas; numerics; hexdigits; specials
alphas     <A>;...;<Z>;\
           <a>;...;<z>;\

\d192;...;\d214;\d216;...;\d246;\d248;...;\d255
numerics   <0>;<1>;\d050;\x33;\
           <4>;...;<9>
hexdigits  <0>;...;<9>;\
           <a>;...;<f>;\
           <A>;...;<F>;

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);(\d246,\d214);\
(\d252,\d220)
END LC_CTYPE
LC_TDMCODESET
ISO88591
END LC_TDMCODESET

```

Collations

A collation is an SQL object that contains rules for collating sequence (the sequence in which characters are ordered for sorting), case (whether characters are uppercase or lowercase), and character class and character string equivalence (whether character variants should be handled as equivalents or whether character variants should be handled as one letter).

NonStop SQL/MP supports collations for single-byte character sets, but not for double-byte character sets. (Double-byte character values always collate in binary order and cannot be upshifted.)

When you create or index a column that has a character data type and a single-byte character set, you can specify the name of a collation to associate with the column. The collation defines the default sort order for values in the column within the table or index. When you create a column that is part of the primary key for the table and associate a collation with that column, the collation also affects the storage order for rows in the table.

You can also specify a collation as part of a character expression that uses single-byte character strings (for example, an expression that compares two character strings from the ISO88591 character set), modifying the ordering and equivalence relationships that determine the result of the expression. (For more information, see [Character Expressions](#) on page C-14.)

A collation name must be a Guardian name.

You create a collation with the CREATE COLLATION statement.

You associate an existing collation with a column or index when you specify the data type for the column or index at the time you create it with the CREATE TABLE, ALTER TABLE, or CREATE INDEX statement. For information on the clause that specifies the data type on these statements, see [Data Types](#) on page D-1.

NonStop SQL/MP includes a set of Guardian procedures that you can invoke from host language programs to compare collations or return information about a collation. For more information about these procedures, see the *SQL/MP Programming Manual for COBOL* or the *SQL/MP Programming Manual for C*.

Column Identifier

A column identifier is used in some SQLCI report writer statements to specify a column in the result of a SELECT command or a named column in the detail list.

```
{ column-name }
{ COL number }
{ alias       }
{ detail-alias }
```

column-name

is the name of a column specified in the select list. You must qualify an unqualified name if it is the same as any other unqualified name in the select list. For example, if the select list includes EMPLOYEE.DEPTNUM and DEPT.DEPTNUM, you must qualify these names when specifying a column identifier. If only EMPLOYEE.DEPTNUM is in the select list, you can omit the qualifier.

COL number

specifies the column in position *number* of the select list. The first item in the select list is COL 1. You can use this form to refer to literals and expressions.

alias

is defined in a NAME command. You can use this form to refer to a name you assign to a literal or expression. For more information, see [Alias](#) on page A-6.

detail-alias

is a detail alias name defined in the NAME clause of a DETAIL command. You can use a detail alias name as a column identifier in any command except DETAIL. For more information, see [Detail Alias](#) on page D-46.

Example—Column Identifiers

These are different ways to designate the same column:

```
SELECT DEPTNUM FROM $SQL.PERSNL.EMPLOYEE ;
SELECT EMPLOYEE.DEPTNUM FROM $SQL.PERSNL.EMPLOYEE ;
SELECT DNUM FROM $SQL.PERSNL.EMPLOYEE ;
```


Columns

A column is a vertical component of a table, the relational representation of a field in a record. A column contains one data value for each row of the table.

Each SQL column has a name that is an SQL identifier that is unique within the table or view that contains the column.

A qualified column name is a column name qualified by the name of the table or view to which the column belongs, or by a correlation name. If a query refers to columns that have the same name but belong to different tables or views, you must use a qualified column name to refer to the columns within the query. The syntax of a qualified column name is:

```
{ table-name  
  view-name  
  correlation-name } .column-name
```

If you define a correlation name for a column in the FROM clause of a statement, you must use that correlation name (called the “explicit correlation name”) if you need to qualify the column name within the statement.

If you do not define an explicit correlation name in the FROM clause, you can qualify the column name with the name of the table or view that contains the column (called the “implicit correlation name”). You can also use the name of a DEFINE that contains the name of the table or view that contains the column as a qualifier, but you must omit the equals sign (=) that normally precedes the DEFINE name.

You must also refer to a column by a qualified column name if you join a table with itself within a query to compare one row of the table with other rows in the same table.

COLUMNS Table

The COLUMNS table is a catalog table that describes the columns of the tables in the TABLES catalog table. [Table C-3](#) describes the contents of the COLUMNS table.

Table C-3. The COLUMNS Table (page 1 of 2)

Column Name	Data Type	Description
1 TABLENAME *	CHAR(34)	Name of table that contains the column
2 COLNUMBER *	SMALLINT UNSIGNED	Position of column in row (first column is 0)
3 COLNAME	CHAR(30)	Column name
4 COLCLASS	CHAR(1)	S if SYSKEY U if user-defined column
5 DATATYPE	CHAR(18)	Data type of column DATETIME for all date-time types FLOAT for all real types
6 FSDATATYPE	SMALLINT SIGNED	File system data type of column (system use only)
7 COLSIZE	SMALLINT SIGNED	Byte length of data in column
8 SCALE	SMALLINT SIGNED	Scale factor if column is numeric; fractional seconds precision if column is date-time or INTERVAL
9 PRECISION	SMALLINT SIGNED	Number of digits if column is numeric; number of digits of binary precision if column is FLOAT; leading field precision if column is INTERVAL
10 OFFSET	SMALLINT SIGNED	Reserved for internal use by HP
11 UNIQUEENTRYCOUNT	LARGEINT SIGNED	Number of unique entries in column for table; set by UPDATE STATISTICS
12 SECONDHIGHVALUE	VARCHAR(20)	First 20 bytes of second-highest value in column (ignoring nulls); stores numerics in ASCII with an appropriate scale (for example, stores -50,000 as -50 scale 3); date-time items use local civil time; set by UPDATE STATISTICS
13 SECONDLLOWVALUE	VARCHAR(20)	First 20 bytes of second-lowest value in column (ignoring nulls); stored as for SECONDHIGHVALUE; set by UPDATE STATISTICS
14 NULLALLOWED	CHAR(1)	Y if null values allowed N if not

Table C-3. The COLUMNS Table (page 2 of 2)

Column Name	Data Type	Description
15 DEFAULTCLASS	CHAR(1)	S if column has system-defined default U if user-defined default N if no default D if null default
16 DEFAULTVALUE	VARCHAR(36)	Default for column; stored in ASCII if numeric; blank if system-defined primary key; NULL if null; CURRENT DATETIME if date-time type and default SYSTEM
17 PICTURETEXT	VARCHAR(64)	Corresponding picture of column if COBOL used to define column; blank for date-time or real types
18 DATETIMESTARTFIELD	SMALLINT SIGNED	Starting field of date-time or INTERVAL data type: 1 if year 2 if month 3 if day 4 if hour 5 if minute 6 if second 7 if fraction 0 for other types
19 DATETIMEENDFIELD	SMALLINT SIGNED	Ending field of date-time or INTERVAL type; same as DATETIMESTARTFIELD
20 DATETIMEQUALIFIER	VARCHAR(28)	Textual representation of DATETIMESTARTFIELD and DATETIMEENDFIELD; blank for other types
21 UPSHIFT	CHAR(1)	Y if UPSHIFT specified N if not
22 HEADING	CHAR(1)	Y if heading specified N if not
23 HEADINGTEXT	VARCHAR (132)	Heading string if heading specified; " " if not
24 CPRULESNAME	CHAR(34)	Name of collation associated with column; "" if none
25 CPARRAYENTRY	SMALLINT SIGNED	Reserved for internal use by HP
26 CHARACTERSET	CHAR(30)	Name of the character set associated with column

* Indicates primary key

The columns TABLENAME through PICTURETEXT (1 through 17) were created in version 1. The columns DATETIMESTARTFIELD through HEADINGTEXT (18 through

23) were added in version 2, and the columns CPRULESNAME through CHARACTERSET (24 through 26) were added in version 300.

The COLUMNS table has a set of column entries for each partition of a partitioned table.

All CHAR and VARCHAR fields in the COLUMNS table except for HEADING and HEADINGTEXT use uppercase characters. Guardian names in the table are fully qualified.

COMMENT Statement

COMMENT is a DDL statement that writes a comment about a collation, column, constraint, index, table, or view to the catalog. For partitioned objects, COMMENT adds the comment for each partition. COMMENT can add a new comment or delete, replace, or add to existing comments.

```
COMMENT ON { COLLATION collation
            { COLUMN column ON { table | view }
            CONSTRAINT constraint ON table
            INDEX index
            TABLE table
            VIEW view
            }
IS comment [ CLEAR ]
```

comment

is the comment, expressed as a string of single-byte or double-byte characters enclosed in single or double quotation marks. It can be 0 to 132 bytes long.

CLEAR

purges existing comments before adding the new comment.

The other clauses identify the object to which the comment applies.

Considerations—COMMENT

- To use COMMENT on a collation, column, table, or view you must be a generalized owner of the table or view. To use COMMENT on an index, you must be a generalized owner of the underlying table. COMMENT also requires authority to write to the affected catalogs.

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 execute a COMMENT statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. (For more information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.)

- To delete a comment, replace all existing comments with a comment that consists of an empty string. For example:

```
COMMENT ON VIEW PERSNL.MGRLIST IS "" CLEAR;
```

- SQL stores comments as rows in the COMMENTS catalog table. You access them by querying the table. Each new comment for an object is stored as a row with a sequence number one greater than the highest existing sequence number for a comment on that object.

An object can have 10,000 comments.

Examples—COMMENT

- This statement adds a comment about a constraint:

```
COMMENT ON CONSTRAINT DATE_ASSRTN ON SALES.ORDERS
IS "Dates are stored as yymmdd.";
```

- This statement replaces all comments on a table with a new comment:

```
COMMENT ON TABLE INVENT.PARTLOC
IS "This table is partitioned LOC_CODE."
CLEAR;
```

- This statement displays comments and help text for a table:

```
SELECT * FROM SALES.COMMENTS
WHERE OBJNAME = "\SYS1.$VOL1.SALES.PARTS";
```

Comments

You can include comments in SQL catalogs, in SQLCI input lines, or in embedded SQL lines.

To add or delete comments about an SQL object from an SQL catalog, use the **COMMENT** statement, which is described in a separate entry.

To indicate that an SQLCI line or an embedded SQL line is a comment, precede the comment with two hyphens (--):

```
-- comment
```

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

Example—Comments

Comments are useful in SQLCI input lines if you use command files that contain SQLCI input. This example shows SQLCI output that echoes comments included in a command file executed with the **OBEY** command:

```
>>OBEY TEST5;
>>-- This command file runs test number 5.
>>  VOLUME TESTDB;  -- Move to subvolume with test programs.
>>-- Set up DEFINES for test
>>  SET DEFMODE ON;
>>  DELETE=*;      -- Delete all current DEFINES.
>>  ADD DEFINE =A, FILE $V10.TEST5.FILEA;
>>  ADD DEFINE =B, FILE $V10.TEST5.FILEB;
>>  ...
```

COMMENTS Table

The COMMENTS table is a catalog table that stores comments and help text for objects defined in the catalog. Each comment or help text line is a row of the table. [Table C-4](#) describes the contents of the COMMENTS table.

Table C-4. The COMMENTS Table

Column Name	Data Type	Description
1 OBJNAME *	CHAR(34)	Name of object commented on; for a constraint, name of table with constraint
2 OBJSUBNAME *	CHAR(30)	Name of column if object type is CL or HC; name of constraint if object type is CN; otherwise ""
3 OBJTYPE *	CHAR(2)	CL if column CN if constraint CP if collation HC if help text for column IN if index TA if table VI if view
4 SEQNUMBER *	SMALLINT UNSIGNED	Line number of portion of this comment
5 COMMENTTEXT	VARCHAR(132)	Line of comment text

* Indicates primary key

COMMENTS was created in version 1 and no new columns have been added in subsequent versions. HC was added to the list of values for column OBJTYPE in version 2; CP was added in version 300.

Guardian names in the COMMENTS table are fully qualified and use uppercase characters. Names in the OBJSUBNAME column also use uppercase characters.

COMMIT Option

COMMIT is an option available on some potentially long-running DDL statements that specifies the start time, the timeout period for lock requests, and the handling of retryable errors for the final phase of the operation.

COMMIT also includes a ROLLBACK option that directs SQL to cancel changes to the database and terminate the operation instead of proceeding with the final phase.

COMMIT is also an option on the CONTINUE statement. CONTINUE continues a DDL operation that specifies COMMIT BY REQUEST and that is ready to enter its final phase.

The final phase of a DDL operation (called the commit phase) always requires exclusive locks on the objects involved. Using the COMMIT option to control the start time, timeout period, and error handling for the final phase can minimize the unavailability of applications that use the objects.

```
{
    {
        [ WHEN READY ]
    }
    COMMIT [WORK] [ [ BY ] REQUEST ] [ on-error ]
    {
        {
            [ AFTER time ]
            [ BEFORE time ]
        }
    }
    ROLLBACK [WORK]
}
```

time is:

```
{
    { mmmbddbyyyy [ , hh:nn ] }
    { ddbmmmbyyyy [ , hh:nn ] }
    { hh:nn }
}
```

on-error is:

```
[
    { value [ SECOND[S] ] } ]
[ TIMEOUT { DEFAULT [ SECOND[S] ] } ]
[ NEVER ] ]
```

[ONCOMMITERROR *commit-option*]

COMMIT [WORK]

directs SQL to begin the commit phase as specified by the other options. If you do not specify other options, SQL uses:

```
COMMIT WORK WHEN READY
TIMEOUT DEFAULT
ONCOMMITERROR ROLLBACK WORK
```


WHEN READY

directs SQL to begin the commit phase as soon as the operation is ready to do so.

[BY] REQUEST

directs SQL to return warning 1619 when the operation is ready to commit and then maintain its ready-to-commit state by performing audit fix-up work as needed until the user responds with a **CONTINUE** statement that specifies a new **COMMIT** option.

Unless you specified the **COMMIT BY REQUEST** in effect in an **ONCOMMITERROR** option, SQL returns warning 1618 just before warning 1619. If you specified the **COMMIT BY REQUEST** in effect in an **ONCOMMITERROR** option, SQL returns the error that activated the **ONCOMMITERROR** option just before warning 1619.

See the examples that follow for the text of the warnings.

For more information about user responses, see [CONTINUE Statement](#) on page C-70.

```
{ | AFTER time | }
```

```
{ | BEFORE time | }
```

specifies that the operation start the commit phase only during the specified time period in local civil time.

If the operation is ready to commit before **AFTER *time***, the operation remains ready to commit until **AFTER *time***, performing audit fix-up work as needed to remain ready to commit. If the operation becomes ready to commit after **BEFORE *time***, an error occurs and the operation performs the action specified in the **ONCOMMITERROR** option.

AFTER *time* can be a time in the past but cannot be more than ten days in the future.

BEFORE *time* must be a time in the future. If you specify both **BEFORE *time*** and **AFTER *time***, **BEFORE *time*** must be greater than **AFTER *time***.

Specify time by using these values as indicated in the syntax diagram:

- b** is a required space.
- mmm** is a 3-character month value (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC).
- dd** is a 2-digit day value (01, 02, ... , 31).

yyyy is a 4-digit year value.

hh is a 2-digit hour value (00, 01, ... , 23).

nn is a 2-digit minute value (00, 01, ... , 59).

If you specify a date but omit time, the time 00:00 is used.

If you specify a time with no date, the current date is used.

The default for AFTER *time* is the current date and time.

The default for BEFORE *time* is 00:00:2100.

```
TIMEOUT { value [ SECOND[S] ] }
        { DEFAULT [ SECOND[S] ] }
        { NEVER }
```

sets the time allowed for lock requests to complete during the commit phase:

value Wait the specified number of seconds (a number in the range 0.01 to 21474836.47); wait indefinitely if *value* is -1.

DEFAULT Wait 60 seconds.

NEVER Wait indefinitely. Your operation acquires the lock after all concurrent locking transactions complete.

The default is TIMEOUT DEFAULT.

If the file system cannot grant a lock request from the operation within the specified time, an error occurs and the operation performs the action specified in the ONCOMMITERROR option.

ONCOMMITERROR *commit-option*

specifies a COMMIT option to take effect if a retryable error occurs during the commit phase.

Retryable errors include file in use, lock request timeouts, resource unavailability, and BEFORE/AFTER time window misses (error numbers 12, 40, 73, 1057, 1615, 1616, 1617, 1618, 1621, 1622, and 3001 to 3999).

A nonretryable error always causes SQL to cancel changes to the database and terminate the operation, regardless what you specify in the ONCOMMITERROR option.

ONCOMMITERROR is recursive because it appears within a COMMIT option and specifies another COMMIT option. You can specify up to three COMMIT options on a single statement; specifying four or more causes an error.

The default is ONCOMMITERROR ROLLBACK WORK.

ROLLBACK [WORK]

cancels changes made to the database during the operation and terminates the operation.

Considerations—COMMIT Option

- Each COMMIT option completely replaces the previous one in effect for the operation. For example, if you execute a DDL statement that includes these options:

```
... WITH SHARED ACCESS NAME OP1
```

```
COMMIT BY REQUEST ONCOMMITERROR COMMIT BY REQUEST;
```

and later continue the operation with this statement:

```
CONTINUE OP1 COMMIT;
```

the ONCOMMITERROR option for the final phase of the operation is the default, ONCOMMITERROR ROLLBACK WORK.

Note. You should normally specify ONCOMMITERROR COMMIT BY REQUEST on the last commit option you specify for an operation, because that protects the operation from automatically being rolled back if a retryable error occurs.

- For information on the phases of an operation that use the COMMIT options, see [WITH SHARED ACCESS OPTION](#) on page W-4.

Example—COMMIT Option

This example shows an ALTER TABLE operation begun from SQLCI with a COMMIT BY REQUEST option. When the operation is ready to begin its final phase (possibly long after the user enters the initial ALTER TABLE statement), the user issues a CONTINUE statement that includes different COMMIT options.

COMMIT AFTER 02:00 specifies a start time of 2:00 am, when there is unlikely to be significant activity on the objects involved. TIMEOUT NEVER directs the operation to wait indefinitely for its lock requests to complete.

The CONTINUE statement also specifies ONCOMMITERROR COMMIT BY REQUEST so that SQL does not automatically roll back the operation if a retryable error occurs during the final phase. This specification gives the user an opportunity to fix problems that cause the error and continue the operation without restarting from the beginning. (The output in the example indicates that the operation terminates without error.)

```
>>ALTER TABLE $HR.PERSONEL.EMP MOVE TO $HDQ.PERSONEL.EMP
+> WITH SHARED ACCESS COMMIT BY REQUEST;
```

```

*** WARNING from SQL [1618]: The ALTER_TABLE statement
    is ready to commit.

*** WARNING from SQL [1619]: To continue processing, please
    enter a commit or rollback with a CONTINUE command.

D>

```

```

.      The user can respond immediately or enter
.      other commands first. (See
.      CONTINUE Statement on page C-70
.      for restrictions on other commands.)

```

```
D>CONTINUE ALTER_TABLE COMMIT AFTER 02:00
```

```
+> TIMEOUT NEVER
```

```
+> ONCOMMITERROR COMMIT BY REQUEST;
```

```

.
.      Time passes until after 2:00 am,
.      when SQL confirms the operation.

```

```
--- SQL operation complete.
```

```
>>
```

Notice that SQLCI prompts with “D>” (the dedicated-operation-in-progress prompt), rather than the usual “>>”, while the ALTER TABLE operation executes in the background, resumes the normal prompt after the operation finishes.

COMMIT WORK Statement

COMMIT WORK is a transaction control statement that ends a TMF transaction and commits any changes to audited objects made during the transaction. COMMIT WORK releases locks on audited and (unless you specify the AUDITONLY option) nonaudited objects.

COMMIT WORK [AUDITONLY]

AUDITONLY

directs SQL to retain existing locks on nonaudited objects.

If your program holds locks on nonaudited objects and specifies AUDITONLY, the program must explicitly use CLOSE, UNLOCK TABLE, or FREE RESOURCES to close cursors and release locks.

In case of audited tables, COMMIT WORK AUDITONLY does not release locks on these audited tables if any of the following conditions is true:

- The plan is SERIAL and it does not have a SORT operation but had access type set to BROWSE ACCESS.
- The plan is SERIAL and it has a SORT operation (irrespective of the access type).
- The plan is PARALLEL (irrespective of the sort and access type).

To unlock all audited tables in any condition, ADD a DEFINE named `=_SQL_AUDITONLY_CLOSE_CR` and run the `COMMIT WORK AUDITONLY` command again.

Considerations—COMMIT WORK

- `BEGIN WORK` starts a transaction. `COMMIT WORK` or `ROLLBACK WORK` ends a transaction. For more information, see [TMF Transactions](#) on page T-6.
- `COMMIT WORK` does not cause a write to disk.
- Within a host program, using `COMMIT WORK` is equivalent to using these sequence of statements:

`FREE RESOURCES` (an SQL statement)

`ENDTRANSACTION` procedure call

`COMMIT WORK` returns status information to the SQLCA so you can use `WHENEVER` to check for error conditions.

Example—COMMIT WORK

Suppose that your application adds information to the inventory. You have just received 24 terminals from a new supplier and want to add the supplier and update the quantity on hand. The part number for the terminals is 6402, and the supplier is assigned supplier number 17. The cost of each terminal is \$800.

The transaction must add the supplier to the `SUPPLIER` table, add the order for terminals to `PARTSUPP`, and update `QTY_ON_HAND` in `PARTLOC`. After the `INSERT` and `UPDATE` statements execute successfully, you commit the transaction.

```
>> VOLUME INVENT;
>> BEGIN WORK;
>> INSERT INTO PARTSUPP VALUES (6402, 17, 800.00, 24);
--- 1 row(s) inserted.
>> INSERT INTO SUPPLIER VALUES (17, "Super Peripherals",
+> "4751 Sanborn Way", "Santa Rosa", "California", "95405");
--- 1 row(s) inserted.
>> UPDATE PARTLOC SET QTY_ON_HAND = QTY_ON_HAND + 24
+> WHERE PARTNUM = 6402 AND LOC_CODE = "G48";
--- 1 row(s) updated.
>> COMMIT WORK;
```

Comparison Predicate

The comparison predicate compares the values of two expressions, the values of two sets of expressions, or the value of an expression and a single value that is the result of a subquery.

```
{ expression comparison-op { expression }
  { subquery }
  { row-value-spec comparison-op row-value-spec }
```

comparison-op is:

```
= Equal
<> Not equal
< Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to
```

row-value-spec is:

```
{ expression [ , expression ] ...
  ( expression [ , expression ] ... ) }
```

Considerations—Comparison Predicate

- The two *row-value-specs* must contain the same number of expressions.

Two *row-value-specs* are equal if all values at the same ordinal position are equal. Their relation is determined by comparing the values in the first ordinal position. If these values are equal, the values in the next ordinal position are compared, and so forth, until the exact relation is determined.

The data type of the first expression must be compatible with the data type of the second expression. The data type of an expression in the first *row-value-spec* must be compatible with the data type of the corresponding expression in the second *row-value-spec*.

The subquery result must be a single value. If the subquery evaluates to more than one row, the comparison results in an error. If no rows satisfy the search condition of the subquery, the predicate evaluates to null.

You cannot use a comparison predicate in a WHERE, ON, or HAVING clause to retrieve all rows where *expression* is null; use the IS NULL predicate instead.

- You can compare two character strings only if both strings are associated with the same character set.

For comparisons between character strings of different lengths, the shorter string is padded on the right with spaces (HEX 20) until it is the length of the longer string.

A HEX 20 is always used for padding, regardless of whether a single-byte or double-byte character set is associated with the expression.

Both fixed-length and variable-length strings are padded in this way. For example, SQL considers the string “JOE” equal to a value JOE stored in a column of data type CHAR or VARCHAR of width three or more. Similarly, SQL considers a value JOE stored in any column of the CHAR data type equal to the value JOE stored in any column of the VARCHAR data type.

Two strings are equal if all characters in the same ordinal position are equal. Lowercase and uppercase letters are not considered equivalent unless used with a collation that equivalences them.

SQL determines collations for character comparisons:

- If neither value is associated with a collation, use binary comparison.
- If only one value is associated with a collation or if both values are associated with the same collation, use that collation.
- If each value is associated with a different collation but one collation is specified implicitly and one collation is specified explicitly, use the collation that is specified explicitly.

You cannot compare character values that are implicitly associated with different collations unless you explicitly specify a collation for the comparison.

You cannot compare character values that are explicitly associated with different collations. SQL returns an error if you attempt to do so.

- For comparisons between numbers, decimal-type numbers are converted to binary. Any exact numeric data type is compatible with all other exact numeric data types. Floating-point data types are sometimes compatible with exact numeric data types. Before evaluation, all values in an expression are first converted to the maximum precision needed anywhere in the expression.
- You can compare only those date-time values with the same range of datetime fields. To compare two date-time values with a different range of fields, use the EXTEND function to expand the values.

For example, to compare a DATETIME DAY TO MINUTE column with a DATETIME YEAR TO HOUR column, expand both values to the range DATETIME YEAR TO MINUTE for the comparison:

```
EXTEND (DATE1, YEAR TO DAY) > EXTEND (DATE2, YEAR TO DAY)
```

For information about the expansion, see [EXTEND Function](#) on page E-28.

- For comparisons of INTERVAL values, SQL first converts the intervals to a common base unit. If no common unit exists, SQL reports an error.
- If a search condition contains a predicate of the form

expression comparison-operator subquery

and the subquery returns no values, the predicate evaluates to null.

For example, this predicate evaluates to null because the subquery returns no value (there is no part number with more than 1500 units in stock):

```
PARTNUM = (SELECT PARTNUM
           FROM ODETAIL
           WHERE QTY_ORDERED > 1500)
```

Examples—Comparison Predicate

- These are some simple comparison predicates:

CUSTNUM = 3210	The customer number is equal to 3210.
----------------	---------------------------------------

SALARY > (SELECT AVG (SALARY) FROM PERSONL.EMPLOYEE)	The salary is greater than the average salary of all employees.
---	---

CUSTNAME = "BACIGALUPI"	The customer name is BACIGALUPI.
-------------------------	----------------------------------

- This comparison predicate evaluates to null for any rows in either CUSTOMER or ORDERS that contain a null value in the CUSTNUM column:

```
CUSTOMER.CUSTNUM > ORDERS.CUSTNUM
```

- This example uses a multivalue comparison predicate to compare multiple values. Use multivalue predicates whenever possible; they are generally more efficient than equivalent search conditions without multivalue predicates.

In this example, the multivalue predicate returns information about anyone whose name follows MOSS, DUNCAN in a list arranged alphabetically by last name and, for the same last name, alphabetically by first name. REEVES, ANNE meets this criteria, but MOSS, ANNE does not.

```
LAST_NAME, FIRST_NAME >= "MOSS" , "DUNCAN"
```

The multivalue predicate is equivalent to this search condition relating three comparison predicates:

```
( LAST_NAME > "MOSS" ) OR
( LAST_NAME = "MOSS" AND FIRST_NAME >= "DUNCAN" )
```

For guidelines about using multivalued predicates, see the *SQL/MP Query Guide*.

- This comparison predicate compares two DATETIME values:

```
EXTEND (TIME1, DAY TO SECOND)
> EXTEND (TIME2, DAY TO SECOND)
```

TIME1, defined as DATETIME HOUR TO SECOND, is 09:06:24.

TIME2, defined as DATETIME DAY TO MINUTE, is 15:09:21.

To evaluate `TIME1 > TIME2`, the range of `DATETIME` fields for each value is extended to a range that includes all the fields from both values.

If the predicate is evaluated on June 16, the `EXTEND` functions return these values, and the comparison predicate returns `TRUE`:

```
16:09:06:24 > 15:09:21:00
```

If the predicate is evaluated on August 15, the `EXTEND` functions return these values, and the comparison predicate returns `FALSE`:

```
15:09:06:24 > 15:09:21:00
```

- This predicate compares two `INTERVAL` values:

```
JOB1_TIME < JOB2_TIME
```

`JOB1_TIME`, defined as `INTERVAL DAY TO MINUTE`, is 2 days 3 hours.

`JOB2_TIME`, defined as `INTERVAL DAY TO HOUR`, is 3 days.

To evaluate the predicate, the SQL converts the two `INTERVAL` values to `MINUTE` and finds the comparison predicate to be true.

- These examples contain a subquery in a comparison predicate. Each subquery operates on a separate logical copy of the `EMPLOYEE` table. The processing sequence is outer to inner. A row selected by an outer query allows an inner query to be evaluated and a single value is returned. The next outer query is evaluated when it receives a value from the inner query.

This query finds all employees whose salary is greater than the maximum salary of department 1500:

```
SELECT FIRST_NAME, LAST_NAME
FROM PERSNL.EMPLOYEE
WHERE SALARY > (SELECT MAX (SALARY)
                FROM PERSNL.EMPLOYEE
                WHERE DEPTNUM = 1500)
```

In this query, the innermost subquery determines the average salary of employees in department 1500. Suppose that the default subvolume is `PERSNL`.

The first outer query of this subquery determines the minimum salary of employees from other departments whose salary is greater than the average salary for department 1500. The main query then finds the names of employees who are not in department 1500 and whose salary is less than the minimum salary determined by the first outer subquery.

```
SELECT FIRST_NAME, LAST_NAME
FROM PERSNL.EMPLOYEE
WHERE DEPTNUM <> 1500 AND
SALARY < (SELECT MIN (SALARY) FROM PERSNL.EMPLOYEE
          WHERE DEPTNUM <> 1500 AND
          SALARY > (SELECT AVG (SALARY) FROM
                    PERSNL.EMPLOYEE
                    WHERE DEPTNUM = 1500) )
```

COMPUTE_TIMESTAMP Function

COMPUTE_TIMESTAMP is an SQLCI function that returns a Julian timestamp in Greenwich mean time for a specified date and time. The data type of the returned value is NUMERIC(18) or LARGEINT.

COMPUTE_TIMESTAMP works in the report writer commands BREAK FOOTING, BREAK TITLE, DETAIL, PAGE FOOTING, PAGE TITLE, REPORT FOOTING, and REPORT TITLE. It also works in the SQLCI commands EXECUTE and SET PARAM. COMPUTE_TIMESTAMP does not work in DML statements or other SQL statements.

<pre> COMPUTE_TIMESTAMP ({mm/dd/yyyy {mm/dd/yyyy hh:nn:ss:mss:uss} {yyyy, mm, dd [, hh, nn, ss, mss, uss]} </pre>	
<i>yyyy</i>	Year (1 through 9999, 1-4 digits)
<i>mm</i>	Month (1 through 12, 1-2 digits)
<i>dd</i>	Day (1 through 31, 1-2 digits)
<i>hh</i>	Hour (0 through 23, 1-2 digits)
<i>nn</i>	Minute (0 through 59, 1-2 digits)
<i>ss</i>	Second (0 through 59, 1-2 digits)
<i>mss</i>	Millisecond (0 through 999, 1-3 digits)
<i>uss</i>	Microsecond (0 through 999, 1-3 digits)

Considerations—COMPUTE_TIMESTAMP

- COMPUTE_TIMESTAMP converts the timestamp from local civil time to Greenwich mean time.
- If you use one of the first two forms of the argument, the argument must appear on a single input line.

For the third form, any of the arguments can be either a numeric literal or a column identifier (but not a numeric expression).

The range of valid dates is from 01/02/0001 00:00:00:000:000 through 12/31/9999 00:00:00:000:000.

- COMPUTE_TIMESTAMP is not compatible with date-time data types. It returns a value of data type NUMERIC(18) or LARGEINT, not data type TIMESTAMP or DATETIME YEAR TO FRACTION.

Example—COMPUTE_TIMESTAMP

This SQLCI command sets the parameter ?D to the Julian timestamp for the date and time in parentheses:

```
SET PARAM ?D COMPUTE_TIMESTAMP (2/8/93 13:25:00:00:00);
```

CONCAT Clause

CONCAT is an SQLCI report writer clause that specifies print items to display without intervening or trailing spaces.

CONCAT works in the BREAK FOOTING, BREAK TITLE, DETAIL, PAGE FOOTING, PAGE TITLE, REPORT FOOTING, and REPORT TITLE report writer commands.

```
CONCAT ( print-list ) [ AS format ]

print-list is:

    print-item [ , print-item ]

print-item is:

    { { { column-id } [ AS format ] } [ STRIP ] }
      { literal
      { arith-expr
      { CONCAT ( print-list ) [AS format]
      { IF cond-expr THEN ( print-list )
        [ ELSE ( print-list ) ]
      { SPACE [ number ]
```

print-item

is an item to concatenate and, optionally, a set of instructions for formatting the item.

Except for the AS, STRIP, and SPACE clauses, which are described in this entry, the descriptions of elements of *print-item* shown in the syntax box are the same as the descriptions for elements of *print-item* in the DETAIL command. For more information, see [DETAIL Command](#) on page D-47.

AS *format*

is an *A_w* or a *C_n* display descriptor that specifies the width of the result of the concatenated print list.

For more information about these descriptors, see [Display Descriptors for Character Items](#) on page A-61. Note that the $C_n.[w]$ format described under AS is not allowed in the CONCAT clause.

STRIP

directs the report writer to strip trailing blanks from the values in the list before concatenating them.

SPACE *number*

specifies the number of spaces between the items in the list. Each space occupies one single-byte print position, regardless of the character set used.

If you omit this clause, the default is 0. (Note that this default is different from the default for the SPACE clause on other statements.) If you specify SPACE but omit *number*, the default is 1.

Considerations—CONCAT

- Items specified in a CONCAT clause have no default heading. You must specify a heading if you want one.
- The default format for a item built from concatenated items is A_n , in which n is the sum of the widths of the concatenated items.

The width of an item from a table is the width specified in the catalog definition for the column that contains the item. The width of a string literal is the number of characters in the string. The report writer estimates and sets a maximum width for the result of an expression.

- You can concatenate single and double-byte characters, but you should avoid concatenations that cause double-byte characters to begin or end in the middle of a word. SQL does not prevent this, but the characters in such strings will be scrambled if you attempt to print them using SQLCI.

Example—CONCAT

This clause concatenates the CITY and STATE values and restricts the formatted result to 25 single-byte characters:

```
CONCAT ( CITY STRIP, ", ", STATE ) AS A25
```

Suppose that a column of the report contains these. Note that the entry for North Carolina is truncated to 25 single-byte characters.

```
AJO, ARIZONA
NEEDLES, CALIFORNIA
WHEAT RIDGE, COLORADO
SECAUCUS, NEW JERSEY
SWEET HOME, OREGON
WINSTON-SALEM, NORTH CARO
```

Concurrency

Concurrency is access to the same data by two or more processes at the same time. The degree of concurrency available (that is, whether a process that requests access to data that is already being accessed is given access or placed in a wait queue) depends on the purpose of the access (read or update), on the access mode, and on whether virtual sequential block buffering (VSBB) is used for the access.

NonStop SQL/MP provides concurrent database access for most operations, controlling access through the locking mechanism and the mechanism for opening and closing tables. For DML operations and for some DDL operations, the access options and locking options you select affect the degree of concurrency. For more information about these options, see [Access Options](#) on page A-1, [Locking](#) on page L-48, and [WITH SHARED ACCESS OPTION](#) on page W-4.

Concurrent access is not possible for all DDL and utility operations, especially those that change timestamps of SQL objects. To maximize concurrency for operations that would not otherwise allow it, NonStop SQL/MP performs certain DDL operations in phases:

- ALTER INDEX, ALTER TABLE, and CREATE INDEX operations that use WITH SHARED ACCESS allow concurrent access by DML statements throughout all but a relatively brief commit phase at the end of the operation.
- ALTER TABLE, ALTER INDEX, CREATE CONSTRAINT, UPDATE STATISTICS, and CREATE INDEX operations without WITH SHARED ACCESS allow concurrent access by DML statements that use SELECT with BROWSE or SHARED access during an initial scan phase, but lock out DML accesses during a later update phase.

These three tables shows the limits on concurrency that are imposed by DDL and utility operations:

- [Table C-5](#) on page C-65 shows DDL operations that limit (or are limited by) DML operations. For DDL operations that occur in two phases, the table shows both scan and update (change timestamp) phases.
- [Table C-6](#) on page C-67 shows the forms of the ALTER statement that change the timestamp of an object. The concurrency of an ALTER operation depends on whether ALTER changes a timestamp.
- [Table C-7](#) on page C-68 shows utility operations that limit concurrency.

Table C-5. Summary of Concurrent DDL and DML Operations (page 1 of 3)

DDL Operations You Can Start	DML Operations in Progress			
	SELECT BROWSE	SELECT SHARED	SELECT EXCLUSIVE	DELETE/INSERT UPDATE
ALTER				

Table C-5. Summary of Concurrent DDL and DML Operations (page 2 of 3)

WITH SHARED ACCESS	A	A	Wait	Wait
Timestamp change	A ¹	Wait	Wait	Wait
No timestamp change	A	A	A	Wait
CREATE CONSTRAINT	A ¹	Wait	Wait	Wait
DEFERRED	A ¹	Wait	Wait	Wait
CREATE INDEX	A ¹	A	Wait	Wait
WITH SHARED ACCESS	A	A	Wait	Wait
UPDATE STATISTICS	A ¹	Wait	Wait	Wait
UPDATE STATISTICS				
Scan phase	A	A	Wait	Wait
Change timestamp	A ¹	Wait	Wait	Wait
DML Operations You Can Start				
DDL Operations in Progress	SELECT BROWSE	SELECT SHARED	SELECT EXCLUSIVE	DELETE/INSERT UPDATE
ALTER				
WITH SHARED ACCESS	A ²	A ²	A ²	A ²
Timestamp change	A ¹	Wait	Wait	Wait
No timestamp chg	A	A	A	A
CREATE CONSTRAINT				
Scan phase	A	A	Wait	Wait
Change timestamp	A ¹	Wait	Wait	Wait
DEFERRED				
Change timestamp	A ¹	Wait	Wait	Wait
Scan phase	A	A	A	A
CREATE INDEX				
WITH SHARED ACCESS	A ²	A ²	A ²	A ²
DML Operations You Can Start				
DDL Operations in Progress	SELECT BROWSE	SELECT SHARED	SELECT EXCLUSIVE	DELETE/INSERT UPDATE

Table C-5. Summary of Concurrent DDL and DML Operations (page 3 of 3)

Scan phase	A	A	Wait	Wait
Change timestamp	A ¹	Wait	Wait	Wait

A Allowed

Wait Started operation waits for the operation in progress to complete. The waiting operation might also time out.

¹ DDL operation aborts the DML operation

² Allowed except during commit phase

Table C-6. ALTER Operation Effects on Timestamps

ALTER Operation	Timestamp Updated	Timestamp Unaffected
ALTER INDEX		
ADD/DROP PARTITION	X	
RENAME		X
File attributes		X
ALTER TABLE		
ADD COLUMN	X	
ADD/DROP PARTITION	X	
COLUMN HEADING		X
File attributes		X
RENAME		X
Security attributes		X
ALTER VIEW		
COLUMN HEADING		X
RENAME		X
Security attributes		X
File attributes and whether you can alter them for tables (T) or indexes (I) are:		
ALLOCATE	T, I	
AUDIT	T	
AUDITCOMPRESS	T, I	
BUFFERED	T, I	
LOCKLENGTH	T, I	
MAXEXTENTS	T, I	
RESETBROKEN	T, I	
SERIALWRITES	T, I	
TABLECODE	T, I	
VERIFIEDWRITES	T, I	

Table C-7. Limits on Concurrent Utility and DML Operations

Utility Operations	DML Operations			
	SELECT BROWSE	SELECT SHARED	SELECT EXCLUSIVE	DELETE/INSERT UPDATE
COPY from a table				
Without SHARE option	A	A	A ¹	N
With SHARE option	A	A	A	A
DUP a table	A	N	N	N
LOAD from a table				
Without SHARE option	A	A	A ¹	N
With SHARE option	A	A	A	A

A Allowed

N Not allowed

¹ Intermittent conflict can occur. If a SELECT locks a row for five minutes, the utility access to the row can time out and abort the utility. Also, a SELECT attempting to access a row can abort if a utility locks the row for longer than the SELECT time out.

Effect of VSBB on Concurrency

NonStop SQL/MP provides virtual sequential block buffering (VSBB) for read, update, and insert operations. Although often more efficient than operations that do not use VSBB, VSBB can cause increased lock waits and timeouts. In general, if you are experiencing concurrency problems, use CONTROL TABLE to disable VSBB. You can use EXPLAIN to determine if SQL is choosing VSBB for your application.

- Sequential read operations

For sequential read operations that use VSBB, the disk process locks all rows within the block (rather than a row at a time). Consequently, SQL operations that use VSBB, even with STABLE access, can acquire locks that remain in place longer than operations that do not use VSBB.

If you are experiencing concurrency problems during read operations, disable VSBB by specifying this directive:

```
CONTROL TABLE * SEQUENTIAL READ OFF
```

- Sequential update operations

For sequential update operations, SQL performs a sequential read before performing the update. Consequently, the same problem occurs as for sequential read operations. If you are experiencing concurrency problems during update operations, disable VSBB by specifying this directive:

```
CONTROL TABLE * SEQUENTIAL UPDATE OFF
```

- Sequential insert operations

For sequential insert operations, the disk process acquires a range protector lock on the row that follows the last row inserted. If the last row inserted is at the end of the file, the range protector lock is placed at the end of the file; consequently, other servers cannot insert rows at the end of the table or view.

For inserts into a key-sequenced table that uses a SYSKEY column or a timestamp as the primary key, VSBB is the usual method for insert operations. If concurrent servers are inserting records into the table, a high percentage of lock waits and timeouts might occur.

If you are experiencing concurrency problems during insert operations, you should disable VSBB by specifying this directive:

```
CONTROL TABLE * SEQUENTIAL INSERT OFF
```

An application designed for NonStop SQL/MP version 1 might experience concurrency problems under version 2 or later because NonStop SQL/MP extended VSBB to update and insert operations in version 2. If a change in concurrency occurs when you move an application from version 1, check VSBB usage.

Constraints

Constraints are SQL objects that help to protect the integrity of data in a table by specifying a condition or conditions that all the values in a particular column of the table must satisfy.

Adding a constraint allows you to determine whether values exist that violate the constraint because SQL rejects the constraint if such values exist. Adding the constraint also keeps such values from being added to the table because SQL rejects any such values after the constraint is in place.

A constraint name is an SQL identifier.

For more information, see [CREATE CONSTRAINT Statement](#) on page C-139.

CONSTRNT Table

The CONSTRNT table is a catalog table that describes the constraints placed on tables. [Table C-8](#) describes the contents of the CONSTRNT table.

Table C-8. The CONSTRNT Table

Column Name	Data Type	Description
1 TABLENAME*	CHAR(34)	Name of table that constraint protects
2 CONSTRAINTNAME*	CHAR(30)	Name of constraint
3 SEQNUMBER	SMALLINT UNSIGNED	Sequence number of constraint on table (for system use only)
4 CONSTRAINTTEXT	VARCHAR (3000)	Text of search condition defining constraint

* Indicates primary key

The CONSTRNT table was created in version 1 and has not been modified in subsequent versions.

Guardian names in the CONSTRNT table are fully qualified. All CHAR and VARCHAR columns in the table use uppercase characters except for lowercase literals specified as part of search conditions stored in the CONSTRAINTTEXT column. Search conditions that include literals that specify the system default multibyte character set are stored as if you specified the actual character set. (For example, if the system default multibyte character set is Kanji, the literal N"" is stored as _KANJI"....".)

CONTINUE Statement

CONTINUE is a DDL statement that specifies a COMMIT option for a DDL operation ready to enter its final phase.

You can execute CONTINUE only after SQL returns warning 1619 ("To continue processing, please enter a commit or rollback with a CONTINUE statement.") to indicate that a DDL operation is ready to enter its final phase. This warning occurs only if the DDL statement that started the operation specified COMMIT BY REQUEST.

```
CONTINUE operation-name { COMMIT [WORK] options }
                        { ROLLBACK [WORK]
```

operation-name

is the name of the operation to continue, as specified by the NAME option in the DDL statement that started the operation.

If you did not specify the NAME option, *operation_name* is based on the type of statement that started the operation (ALTER_TABLE, CREATE_INDEX, and so forth).

```
{ COMMIT [WORK] options }
{ ROLLBACK [WORK] }
```

is a commit option that controls the start time for the final phase of the operation and specifies the timeout period for lock requests and the handling of retryable errors during the commit phase of the operation. The commit option can also direct SQL to cancel changes made by the operation and terminate the operation.

For more information, see [COMMIT Option](#) on page C-52.

Considerations—CONTINUE

- You cannot execute the CONTINUE statement within a user-defined transaction. If you attempt to do so, SQL terminates the operation in effect and rolls back the transaction, canceling all changes made to the database by either the operation or the transaction.
- A host language or SQLCI process that receives warning 1619 must respond with a CONTINUE statement before terminating or taking any action that directly or indirectly modifies a DEFINE. However, the process can execute other SQL statements or SQLCI commands before executing a CONTINUE statement, except for DDL statements or write-access utility operations on source or target objects of the operation in progress.

If the process terminates or modifies a DEFINE before executing a CONTINUE statement, the DDL operation in progress stops without being either committed or canceled. The changes are not made (as they would be if the operation was committed), and new SQL objects created by the operation are not removed (as they would be if the operation was canceled).

If this scenario occurs, you must use CLEANUP to remove the new SQL objects (or ask another user with access to the super ID to do so). If the operation was a CREATE INDEX operation on a table with the AUDITCOMPRESS file attribute, you might need to use ALTER TABLE to reset AUDITCOMPRESS.

An SQLCI process that uses a terminal as an IN file protects you from this situation by not allowing any of these commands (which terminate the SQLCI process or directly or indirectly modify DEFINES) between error 1619 and a CONTINUE statement:

ADD DEFINE	ALTER DEFINE	CATALOG	DELETE DEFINE
EXIT	SET DEFMODE	SYSTEM	VOLUME

The SQLCI process also behaves as if you specified SET BREAK_KEY OFF, returning control to the previous Break key owner (usually TACL) if you press the Break key. (This strategy allows you to return to TACL to execute other commands before continuing the DDL operation, and then type PAUSE to return to SQLCI and execute a CONTINUE statement.) In addition, SQLCI prompts with a dedicated-operation-in-progress prompt (D>) instead of the normal SQLCI prompt to remind you that an operation in progress is waiting for a CONTINUE statement.

An SQLCI process that does not use a terminal as an IN file (for example, one which reads SQLCI commands from a disk file) also protects you from modifying DEFINES but does not protect you from terminating the SQLCI process. Terminating the SQLCI process while an operation in progress is waiting for a CONTINUE statement (either by reaching the end of the IN file or by executing an EXIT command) leaves SQL objects in the inconsistent state described earlier in this consideration.

A host language process does not protect you against either modifying DEFINES or terminating the process unless you explicitly code such protection into the program. Make sure that a host language process that receives warning 1619 responds with a CONTINUE statement before terminating or taking any action that modifies a DEFINE.

Note. SQL does not prohibit you from executing a transaction between receiving warning 1619 and executing a CONTINUE statement, but you must complete the transaction before executing the CONTINUE statement. Attempting to execute a CONTINUE statement within a transaction causes SQL to roll back both the operation you attempted to CONTINUE and any other operations included in the transaction.

Examples—CONTINUE

- This SQLCI example cancels a DDL operation about to begin its final phase:

```
>> ALTER TABLE $D1.MDB.EMP PARTONLY MOVE $D2.MDB.EMP
      WITH SHARED ACCESS NAME MOVE_EMP_TABLE
      COMMIT BY REQUEST;
*** WARNING from SQL [1618]: The MOVE_EMP_TABLE
      statement is ready to commit.
*** WARNING from SQL [1619]: To continue processing, please
      enter a commit or rollback with a CONTINUE statement.
D>CONTINUE MOVE_EMP_TABLE ROLLBACK WORK;
*** ERROR from SQL [-1620]: The ROLLBACK was requested
      as part of the commit criteria. The command has
      been aborted.
>>
```

- This SQLCI example commits a DDL operation about to begin its final phase. The example uses a FUP LISTOPENS command to verify that there is no activity on the partition being moved before issuing a CONTINUE statement.

```
>> ALTER TABLE $D1.MDB.PART1 PARTONLY MOVE $D2.MDB.PART1
      WITH SHARED ACCESS NAME PART1_MOVE
      COMMIT BY REQUEST;
*** WARNING from SQL [1618]: The PART1_MOVE statement is
      ready to commit.
*** WARNING from SQL [1619]: To continue processing, please
      enter a commit or rollback with a CONTINUE statement.
D>FUP LISTOPENS $D1.MDB.PART1;
D>CONTINUE PART1_MOVE COMMIT WHEN READY
      ONCOMMITERROR COMMIT BY REQUEST;
--- SQL operation complete.
>>
```

CONTROL EXECUTOR Directive

CONTROL EXECUTOR is a DCL directive that allows or prohibits parallel execution of queries. Parallel execution can decrease the elapsed time for processing a query.

```
CONTROL EXECUTOR PARALLEL EXECUTION { ON | OFF }
```

ON

executes queries in parallel using multiple SQL executors if parallel execution is possible and efficient.

OFF

executes queries using one SQL executor.

OFF is the default.

Considerations—CONTROL EXECUTOR

- CONTROL EXECUTOR affects the way SQL executes DML statements. It does not affect other types of statements.

In SQLCI, CONTROL EXECUTOR remains in effect until you enter another CONTROL EXECUTOR directive or end the SQLCI session.

In a host language program, other scoping rules might also apply to CONTROL EXECUTOR. For more information, see the SQL/MP programming manual for your host language.

- Specifying ON does not force parallel execution to occur. The optimizer still analyzes the query to determine whether a parallel plan has a lower cost than a serial plan. In addition, under certain conditions parallel execution is inappropriate or not currently supported. These conditions include:
 - The query uses a cursor update. Cursor updates are not supported in parallel because row currency cannot be maintained in a parallel plan.
 - The DML statement includes a UNION operation.
 - None of the tables in the query are partitioned.

To find out if parallel execution is chosen in a particular case, use EXPLAIN.

Example—CONTROL EXECUTOR

This example shows an equijoin operation:

```
CONTROL EXECUTOR PARALLEL EXECUTION ON;
SELECT CUSTOMER.CUSTNUM, CUSTOMER.CUSTNAME
FROM =CUSTOMER, =ORDERS
```

```
WHERE CUSTOMER.CUSTNUM = ORDERS.CUSTNUM
STABLE ACCESS;
```

CONTROL QUERY Directive

CONTROL QUERY is an SQL compiler directive that controls plans for queries. Options specify whether to resolve names at execution time or at SQL startup time, whether to include hash join algorithms among algorithms considered for executing queries, and whether to optimize query response time for returning a few rows or all rows.

CONTROL QUERY	{	BIND NAMES { AT EXECUTION }		AT STARTUP	}
		HASH JOIN { OFF ENABLE SYSTEM }			
		INTERACTIVE ACCESS { ON OFF }			
		MDAM { ON OFF }			

```
BIND NAMES { AT EXECUTION | AT STARTUP }
```

specifies when to resolve names of SQL objects and catalogs in DDL, DML, and DSL (GET) statements and in DCL statements other than CONTROL TABLE:

- AT EXECUTION Resolves names just before each statement executes
- AT STARTUP Resolves names just before the first SQL statement of the program executes (static SQL) or when the PREPARE or EXECUTE IMMEDIATE executes (SQLCI and dynamic SQL)

The default is BIND NAMES AT STARTUP.

For a cursor, the execution time for the OPEN statement is considered the execution time for the SELECT statement within the DECLARE CURSOR statement. (Neither DECLARE CURSOR, FETCH, nor CLOSE causes name resolution.)

BIND NAMES AT EXECUTION can lead to unnecessary automatic recompilations unless you specify the CHECK INOPERABLE PLANS option when you explicitly compile the program. For more information about name resolution and its relationship to compilation options, see the SQL/MP programming manual for your host language.

Note that BIND NAMES does not affect the host compiler directive INVOKE or the DCL statement CONTROL TABLE.

`HASH JOIN { OFF | ENABLE | SYSTEM }`

specifies whether to allow SQL to use hash joins (joining algorithms based around hash tables built largely in memory) when the optimizer expects such joins to improve query performance.

<code>OFF</code>	Prohibits the use of hash joins
<code>ENABLE</code>	Allows the optimizer to use hash joins
<code>SYSTEM</code>	Currently allows the optimizer to use hash joins (the same as <code>ENABLE</code>) but is subject to change in future releases

The default for a program or SQLCI session is `HASH JOIN SYSTEM`. Specify `HASH JOIN ENABLE` instead if you want to guarantee that the optimizer always considers hash joins for your queries, even if HP changes the meaning of the `SYSTEM` option.

You should normally leave `HASH JOIN` set to `SYSTEM` or `ENABLE`, because the optimizer is designed to select a hash join only if the resulting plan improves the performance of a query; prohibiting hash joins can degrade performance. You might want to use `HASH JOIN OFF` when you know that contention for memory is unusually severe in the processor or processors in which your queries will run.

The `CONTROL TABLE` directive includes a `JOIN METHOD HASH` option you can use to force the optimizer to select a hash join in specific situations. In such cases, a hash join forced by the `CONTROL TABLE` directive overrides any `CONTROL QUERY HASH JOIN OFF` in effect at the same time.

`INTERACTIVE ACCESS { ON | OFF }`

specifies whether to optimize response time for returning only the first few rows found (`ON`), or for returning all the rows found (`OFF`).

The default for a program or an SQLCI session is `INTERACTIVE ACCESS OFF`.

`INTERACTIVE ACCESS ON` is typically used when you want only the first few rows in a result and you know an index is available on a column. It directs the optimizer to use the index.

`INTERACTIVE ACCESS ON` might not change the optimal plan if a sort is unavoidable. SQL might perform a sort if a query contains a `UNION` operator, `DISTINCT` clause, aggregate function, `ORDER BY` clause, or `GROUP BY` clause. (To avoid a sort, you can create an index on the appropriate columns, such as the columns specified in the `ORDER BY` clause.)

`MDAM { ON | OFF }`

specifies whether or not to consider the Multidimensional Access Method (MDAM) for the query.

`MDAM ON` is the default; it directs SQL to consider using MDAM for the query. `MDAM OFF` turns MDAM off for all tables in the query plan.

If you define a table with a clustering key and use a protection view, you will not be able to use MDAM. For more information, see [Protection View](#) on page P-32.

Considerations—CONTROL QUERY

Static control statements affect only those SQL statements that follow them and are located within the same scope. For example, in the global area or a specific procedure.

Dynamic control statements have temporal scoping. For example, they apply to all dynamic SQL statements that are prepared by any procedure in the entire program after the dynamic control statement is executed. The only exception to this rule is a dynamic CONTROL TABLE TIMEOUT statement that effects on all static statements in the program, and on all dynamic control statements that are prepared before and after the execution of the dynamic CONTROL TABLE TIMEOUT statement.

Examples—CONTROL QUERY

- This directive tells SQL to bind names in subsequent statements at execution time and prohibits the use of hash joins for subsequent queries:

```
CONTROL QUERY BIND NAMES AT EXECUTION HASH JOIN OFF;
```

- This directive ensures that SQL considers hash joins for subsequent queries even if HP changes the default behavior regarding the use of hash joins:

```
CONTROL QUERY HASH JOIN ENABLE;
```

- This directive tells SQL to optimize subsequent queries for returning the first rows in the result:

```
CONTROL QUERY INTERACTIVE ACCESS ON;
```

- This directive tells SQL to enable MDAM for the query:

```
CONTROL QUERY MDAM ON;
```

- This directive resets all CONTROL QUERY options to the default state:

```
CONTROL QUERY HASH JOIN SYSTEM INTERACTIVE ACCESS OFF  
BIND NAMES AT STARTUP;
```


CONTROL TABLE Directive

CONTROL TABLE is a DCL directive that specifies performance-related options for DML accesses to a table or view. CONTROL TABLE affects decisions the SQL compiler makes about how to execute DML statements.

CONTROL TABLE affects the selection of access paths, join methods, join sequences, lock types, and block buffering and block splitting algorithms. CONTROL TABLE also specifies whether to open indexes and partitions at the initial access to a table, whether to checkpoint unaudited writes, and what to do when encountering locked data or unavailable partitions.

```

CONTROL TABLE { table [ AS cor ]
                { view  [ AS cor ] [BASETABLE btcor] }
                { * }
                [
[ ACCESS PATH { SYSTEM | PRIMARY | INDEX index }
[   [MDAM ON [USE {value|DEFAULT} [KEY] COLUMN[S]]
[       [ACCESS { SPARSE | DENSE | SYSTEM}] ]
[
[ JOIN METHOD { NESTED | [KEY SEQUENCED] MERGE |
[   HASH | SYSTEM }
[
[ JOIN SEQUENCE { SYSTEM | sequence-number }
[
[ MDAM { OFF | ENABLE }
[
[ OPEN { ALL | ACCESSED } [ PARTITIONS ]
[
[   { RETURN | WAIT } IF LOCKED
[
[ SEQUENTIAL [ INSERT|READ|UPDATE ] {ON|OFF|ENABLE}
[
[ SEQUENTIAL BLOCKSPLIT [ FOR INSERT ] {ON|ENABLE}
[
[   { SKIP | STOP AT } UNAVAILABLE PARTITION
[
[ SYNCDEPTH { 0 | 1 }
[
[ TABLELOCK { OFF | ON | ENABLE }
[
[ TIMEOUT { value | DEFAULT } [ SECOND[S] ]
[

```

table

is the name of the table to which the control options apply (or an equivalent DEFINE), exactly as the table name (or DEFINE name) appears in subsequent references that are to be affected by the control options.

For example, if *table* is a fully qualified Guardian name, the control options apply to subsequent references that use the same fully qualified Guardian name. The

control options do not apply to references that use only the table name (even if the name expands to the same fully qualified name) nor to references that use a DEFINE name (even if the table associated with the DEFINE is the same table).

Similarly, if *table* is a DEFINE name, the control options apply to subsequent references that use the same DEFINE name. The control options do not apply to references that specify a table name, even if the DEFINE value is the table name.

AS cor

specifies that the control options apply only to instances of the table or view with the correlation name *cor*. The correlation name can be either implicit or explicit.

view

is the name of a view to which the control options apply (or an equivalent DEFINE), exactly as the view name (or DEFINE name) appears in subsequent references that are to be affected by the control options.

For example, if *view* is a fully qualified Guardian name, the control options apply to subsequent references that use the same fully qualified Guardian name. The control options do not apply to references that use only the view name (even if the name expands to the same fully qualified name) nor to references that use a DEFINE name (even if the view associated with the DEFINE is the same view).

Similarly, if *view* is a DEFINE name, the control options apply to subsequent references that use the same DEFINE name. The control options do not apply to references that specify a view name, even if the DEFINE value is the view name.

To apply a CONTROL TABLE statement to one or more tables within a view, you must use either an explicit or implicit correlation name for the table. SQL returns a syntax error if you specify any other type of name.

BASETABLE btcor

specifies that the control options for *view* apply only to the table or tables inside the view that have the correlation name *btcor*. The correlation name can be either implicit or explicit.

*

specifies that the control options apply to all tables referenced in the host language program or SQLCI session. (For specific rules about the scope of the CONTROL TABLE directive in a host program, see the SQL/MP programming manual for the host language used in the program.)

```
ACCESS PATH { SYSTEM | PRIMARY | INDEX index }

[ MDAM ON [ USE {value|DEFAULT} [KEY] COLUMN[S] ]

      [ ACCESS { SPARED | DENSE| SYSTEM} ] ]
```

controls the access path for a DELETE, SELECT, or UPDATE, or for the SELECT portion of an INSERT-SELECT.

SYSTEM directs SQL to choose the access path
PRIMARY specifies the primary access path for the table
INDEX specifies *index* as the access path

The default is ACCESS PATH SYSTEM.

To specify different access paths for different occurrences of the same table within a single query, use correlation names to distinguish occurrences of the table and use the AS *cor* clause in CONTROL TABLE directives that specify access paths.

If you specify the INDEX option and *index* does not exist, SQL issues an error (for SQLCI or dynamic SQL) or warning (for static SQL) when it compiles the CONTROL TABLE statement. If *index* exists, but is not an index for the specified table, SQL issues an error message later when it compiles a DML statement that references the table.

The MDAM ON option directs SQL to use MDAM for the specified table and access path. When you specify MDAM ON, you must specify the PRIMARY or INDEX *index* option; MDAM ON is not supported for ACCESS PATH SYSTEM. You must specify MDAM ON for a specific table; you cannot use this option with CONTROL TABLE *.

You can specify one or both of these optional clauses with MDAM ON:

USE	communicates to SQL how many columns of the key should be used by MDAM.
<i>value</i>	<p>specifies the number of columns to be used by MDAM. <i>value</i> must be an integer than zero. If <i>value</i> exceeds the actual number of key columns defined for the index, SQL chooses the maximum number of key columns that can be used by MDAM.</p> <p>To direct SQL to use the first three columns of an index, specify this: USE 3 KEY COLUMNS</p> <p>The word KEY can be included for clarity, but has no effect.</p>
DEFAULT	directs SQL to determine the number of key columns to be used for MDAM. You can use DEFAULT to reset a previously-specified USE value.
ACCESS	specifies whether a dense or sparse algorithm (or system-determined algorithm) should be used when accessing columns with MDAM, as follows:

USE	communicates to SQL how many columns of the key should be used by MDAM.
DENSE	specifies an adaptive dense algorithm for all columns unless SQL determines that a dense algorithm is not appropriate, as with character or float data types. DENSE is the preferred algorithm when column values are generally sequential. SQL increments each value to obtain the next value.
SPARSE	specifies a sparse algorithm. This is the preferred algorithm when column values are not sequential (such as 25, 135, and 500). SQL uses positioning to obtain the next value of the column.
SYSTEM	specifies that SQL determines the type of algorithm for each column. You can use SYSTEM to reset a previously specified ACCESS option.
For more information about access algorithms, see the <i>SQL/MP Query Guide</i> .	

If you specify both USE and ACCESS, the USE option must precede the ACCESS option. Otherwise, SQL returns an error. If you specify MDAM ON along with CONTROL QUERY MDAM OFF, SQL returns an error.

When you specify the ACCESS PATH...MDAM ON option, SQL resets any unspecified options to their default values. If, for example, you change the value of the USE option and want to preserve a previous ACCESS setting, include the appropriate ACCESS clause in your new CONTROL TABLE statement.

If you define a table with a clustering key and use a protection view, you will not be able to use MDAM. For more information, see [Protection View](#) on page P-32.

JOIN METHOD { NESTED | [KEY SEQUENCED] MERGE | HASH | SYSTEM }

specifies whether to use the nested, sort merge, key-sequenced merge, or hash join method when the specified table is the inner table of a join operation. (For descriptions of these join methods, see the *SQL/MP Query Guide*.)

JOIN METHOD SYSTEM is the default. SYSTEM directs SQL to select an appropriate join method for each join of the table or tables specified in the CONTROL TABLE directive.

If a CONTROL QUERY HASH JOIN OFF directive is in effect, JOIN METHOD SYSTEM never selects a hash join. If you explicitly request JOIN METHOD HASH, however, that CONTROL TABLE directive overrides the CONTROL QUERY directive for the specified table.

SQL cannot perform a hash join on a column or index that has an associated collation. If you specify JOIN METHOD HASH in such a case, SQL returns error -6021.

The KEY SEQUENCED MERGE option applies only to candidates for a key-sequenced merge join operation. If the current composite table and the inner table

are not in the same order (such as in a repartitioning parallel plan), SQL returns an error and does not choose the key-sequenced merge join method.

Because JOIN METHOD applies only to the inner table of a join operation, it is ignored for a table that has a join sequence of 1. JOIN METHOD is normally used only with the JOIN SEQUENCE option.

JOIN SEQUENCE { SYSTEM | *sequence-number* }

specifies the sequence in which SQL processes the table for the join.

JOIN SEQUENCE SYSTEM is the default; it directs SQL to determine the sequence.

sequence-number is an integer that specifies the join sequence for the table within a SELECT. For example, if you specify *sequence-number* as 3 for a table, that table is third in the join sequence for a SELECT.

You do not need to specify *sequence-number* for each table in a SELECT; SQL determines the join sequence for tables you do not mention in CONTROL TABLE directives. For example, if you specify *sequence-number* as 3 for a table, but have no CONTROL TABLE directives in effect for other tables, SQL determines the tables that are first and second in the join order, in addition to those that are fourth, fifth, and so forth.

sequence-number must be greater than zero and no greater than the number of tables (or tables with different correlation names) participating in the SELECT. Two tables (or two occurrences of the same table) cannot have the same join sequence in a SELECT.

MDAM { OFF | ENABLE }

specifies whether to enable MDAM or turn it off.

OFF	turns MDAM off for the specified table
ENABLE	enables the use of MDAM for the specified table. This is the default; ENABLE has an effect only if you previously specified MDAM OFF or ACCESS PATH...MDAM ON for the table.

You cannot specify MDAM OFF or MDAM ENABLE with the CONTROL TABLE * command. The MDAM option must be associated with a specific table.

When you specify MDAM OFF or MDAM ENABLE, SQL sets the ACCESS PATH MDAM options to their default values, USE DEFAULT KEY COLUMNS and ACCESS SYSTEM.

If you define a table with a clustering key and use a protection view, you will not be able to use MDAM. For more information, see [Protection View](#) on page P-32.

`OPEN { ALL | ACCESSED } [PARTITIONS]`

specifies whether to defer opening indexes and remaining partitions in a table until access to the objects is required:

- `ALL` opens all indexes and partitions the first time any partition is accessed
- `ACCESSED` opens indexes and partitions only as needed (called “on-demand opens”)

The default is `OPEN ACCESSED`.

`{ RETURN | WAIT } IF LOCKED`

specifies action if you attempt to access data with `STABLE` or `REPEATABLE` access and the data is locked by another user. (This option does not apply to catalog tables.)

- `RETURN` returns file-system error 73 (SQL error -8300) to SQLCI or (through `SQLCODE`) the host language program.
- `WAIT` waits for data as specified by the `TIMEOUT` option.

The default is `WAIT IF LOCKED`.

If you specify `RETURN IF LOCKED`, make sure the `TIMEOUT` option is large enough to permit a lock request to be passed to the disk process.

You might want to use `RETURN IF LOCKED` for converted Enscribe applications that used alternate locking mode. The operations are similar.

`SEQUENTIAL [INSERT | READ | UPDATE] { ON | OFF | ENABLE }`

specifies whether the file system should buffer `INSERT`, `READ`, or `UPDATE` operations. Buffering reduces the number of messages between the file system and disk process by a factor equal to the blocking factor.

- `ON` buffer operations when possible
- `OFF` do not buffer operations
- `ENABLE` let SQL decide when to buffer

For audited tables, the default is `ENABLE`. For nonaudited tables, the default is `OFF` for `INSERT` and `UPDATE`; the default is `ENABLE` for `READ`.

Use `OFF` if operations will not be sequential or if a small but common subset of rows will be accessed concurrently by more than one process. Use `ENABLE` if you are unsure whether operations will be sequential or random.

Under certain conditions, SQL might determine whether or not to buffer operations regardless of the option you specify. For more information about this and other aspects of buffering, see the comments on buffered operations under [Considerations—CONTROL TABLE](#) on page C-85.

SEQUENTIAL BLOCKSPLIT [FOR INSERT] { ON | ENABLE }

specifies the method of splitting blocks when an INSERT requires a block split:

ON	splits blocks as if inserts are sequential
ENABLE	selects the block split algorithm depending on whether sequential inserts are detected

The default is SEQUENTIAL BLOCKSPLIT ENABLE.

Note. Using SEQUENTIAL BLOCKSPLIT ON when inserts are not actually sequential and in increasing order by primary key (or when intervening records exists in blocks where inserts occur) can waste substantial disk space.

Use SEQUENTIAL BLOCKSPLIT ENABLE unless you are extremely knowledgeable about the way block splits are handled and absolutely certain that your table will receive a series of sequential inserts that the disk process can not recognize as sequential. (For a discussion of such a case, see information about specifying the sequential blocksplit algorithm on page [C-87](#).)

The SEQUENTIAL BLOCKSPLIT option is similar to the SETMODE 91,3 option available for Enscribe files, except that it applies only to tables and views, not to indexes.

{ SKIP | STOP AT } UNAVAILABLE PARTITION

controls whether SQL continues to process a query when a partition required by the access plan of the query is unavailable.

This option applies to both partitioned tables and partitioned indexes, but affects only the main query in a SELECT statement without an INTO clause. (SQL always stops processing and returns an error when a required partition is unavailable for a subquery, a SELECT statement in the search condition of an UPDATE or DELETE statement, a SELECT with an INTO clause, or any other DML or DDL statement.)

SKIP	If a required partition is unavailable, issue warning 8239 (Partition was skipped), open the next partition, and return the next row that satisfies the search conditions of the query.
STOP AT	If a required partition is unavailable, return an error and stop processing the query.

The default is STOP AT UNAVAILABLE PARTITION.

Any of these conditions make a partition unavailable:

- The volume is not available (error 66)
- The file is bad (error 59)
- No more opens are permitted on the volume (error 61)
- A path or network error occurs (errors 200-255)

For an example, see information about local autonomy under [Considerations—CONTROL TABLE](#) on page C-85.

`SYNCDEPTH { 0 | 1 }`

controls the method of writing to the disk process for nonaudited tables and views:

- 0 prevents the disk process from sending checkpoint messages. Might slightly improve performance but makes modifications less reliable because an error during an update that modifies several rows halts processing of the statement.
- 1 enables retry of a message to the disk process, if necessary. Each time data is written to a disk process, the primary disk process sends a checkpoint message to the backup disk process with a description of the operation. This is the preferred option for nonaudited tables, although it is not as safe as auditing.

The default SYNCDEPTH for a table or protection view is 1.

(For audited tables, or views with audited underlying tables, SYNCDEPTH is always 1; directives to change it are ignored.)

`TABLELOCK { OFF | ON | ENABLE }`

specifies whether to use table locks for subsequently compiled DML statements that access the table or view:

- OFF never use table locks
- ON always use table locks

ENABLE SQL decides whether to use table locks.

TABLELOCK ENABLE is the default.

If you want to increase access performance and are not concerned with concurrency, use TABLELOCK ON.

If TABLELOCK OFF is in effect and SQLCI or a host language program attempts to acquire more row locks than allowed, the file system issues error 35 (Lock limit has been reached).

Note that for nonaudited tables, locking protocol enforced by DP2 is not available. Therefore, if sequential block buffering (either RSBB or VSBB) is used to access a nonaudited table, SQL always acquires a table lock, regardless of the setting of the TABLELOCK option and access option.

`TIMEOUT { value | DEFAULT } [SECOND[S]]`

specifies the number of seconds allowed to complete file-system requests in DML operations.

If the time elapses before the file system can grant a request to lock data, the statement fails and SQL returns file-system error 40 (Operation timed out) or error 73 (File/Record locked). (This option does not apply to catalog tables.)

value waits the specified number of seconds (a number in the range 0.01 to 21474836.47) or wait indefinitely (if *value* is -1)

DEFAULT waits 60 seconds

If users often encounter timeouts, increase the time. A low timeout value can cause the application to function well under light load conditions but not under heavy loads.

Avoid timeouts below 0.25 second because the reliability of the timeout mechanism decreases as *n.nn* approaches 0.01.

Considerations—CONTROL TABLE

- During an SQLCI session, CONTROL TABLE affects access to tables and views from subsequent statements. A particular CONTROL TABLE option remains in effect until you enter another CONTROL TABLE statement that changes it.

There are several ways to reference a table or view in a CONTROL TABLE directive: a fully qualified Guardian name, an unqualified Guardian name, a DEFINE, or a correlation name. The name you use to refer to a table or view in CONTROL TABLE must exactly match the name you use in the subsequent DML statement. For example, a CONTROL TABLE directive for Table T has no effect on a SELECT statement from View V, even when View V is a view that references Table T. Similarly, \$V.SV.TableT is not equivalent to Table T even when Table T expands to \$V.SV.TableT.

A CONTROL TABLE directive that includes at least one option, and that makes a reference to a specific table or view with an AS or Basetable clause, typically does not affect the values of options set previously with a more general reference to that table or view. It also does not affect the values of options set with different specific references to that table or view.

For example, these three directives can coexist if entered in the order shown:

```
CONTROL TABLE A ACCESS PATH INDEX INDEX1;
CONTROL TABLE A AS B ACCESS PATH INDEX INDEX2;
CONTROL TABLE A AS C ACCESS PATH INDEX INDEX3;
```

Accesses to table A from references that use correlation names B or C use INDEX2 or INDEX3, respectively, but other accesses to table A use INDEX1.

The order of the directives shown is significant because a CONTROL TABLE directive with at least one option that makes a general reference to a table or view overrides more specific directives that were entered previously. For example, this sequence of directives causes all accesses to table A to use INDEX1 (even those from references that use correlation names B or C), because the general reference

to table A in the final directive overrides the specific references in the preceding directives:

```
CONTROL TABLE A AS B ACCESS PATH INDEX INDEX2;
CONTROL TABLE A AS C ACCESS PATH INDEX INDEX3;
CONTROL TABLE A ACCESS PATH INDEX INDEX1;
```

In a host language program, specific scoping rules might apply to the use of the CONTROL TABLE statement. For more information, see the SQL/MP programming manual for your host language.

- If you specify the CONTROL TABLE directive without options, all previously specified options for the table or view referenced exactly as specified in the directive revert to their default values.

In this case, a general reference to a single table does not affect more specific references to that table:

```
CONTROL TABLE A;    Clear all options set for table A, but not options set for
                      table A AS B
```

```
CONTROL TABLE *;    Clear all options set for all tables
```

- Normally, you should allow SQL to determine access paths, join methods, and join sequences. SQL does this automatically (basing its choices on stored statistics, assumptions about data distribution, and availability of access paths) unless you use the ACCESS PATH, JOIN METHOD, or JOIN SEQUENCE options in the CONTROL TABLE directive.

Use these options only in specific situations in which you know that SQL's current algorithm does not produce optimal results.

Make sure you are familiar with details of query operations as described in the *SQL/MP Query Guide*, and be sure to restore default values for these options immediately afterwards. For example:

```
CONTROL TABLE * ACCESS PATH SYSTEM;
CONTROL TABLE * JOIN METHOD SYSTEM JOIN SEQUENCE SYSTEM;
```

Also, be aware of these special considerations:

- If you specify an access path, a query does not run unless that access path is available. To allow for alternate paths, code your application to check for errors and specify an alternate path if the normally-preferred path is not available. (SQL automatically considers alternate access paths if you do not specify an access path.)
- Certain errors in the specification of ACCESS PATH, JOIN METHOD, or JOIN SEQUENCE cannot be detected and reported until an affected DML query is compiled. For example, if you erroneously specify a JOIN SEQUENCE greater than the number of tables in the next SELECT that includes the table, an error occurs in response to the statement that contains the SELECT, not to the CONTROL TABLE directive.

- If an index for an UPDATE includes a column being updated, such as

```
UPDATE table SET index-column = index-column + 1
```

specifying that index as an access path can cause an operation that never ends (the “Halloween problem” in database literature).

SQL issues an error message if it compiles an UPDATE statement that can lead to the Halloween problem and a CONTROL TABLE directive prevents it from selecting an alternate access path.

EXPLAIN reports indicate whether ACCESS PATH, JOIN METHOD, and JOIN SEQUENCE were forced by the user rather than determined by SQL.

- Normally, the disk process automatically optimizes for sequential or nonsequential inserts by changing its block splitting algorithm for a given table open when it recognizes a series of sequential inserts. In a few unusual cases, however, the disk process fails to recognize sequential inserts.

For example, suppose that each of two separate clients sends a series of sequential records to the same server for insertion in a table. Client A sends records with primary key values 1, 2, 3, and so forth. Client B sends records with primary key values 1001, 1002, 1003, and so forth. The server then sends the interleaved stream of records (1, 1001, 2, 1002, 3, 1003 ...) to the disk process for insertion in the table.

Because the series of records sent by the server is not sequential, the disk process fails to recognize that sequential inserts are occurring (although in two separate sequences within the table) and does not change the block splitting algorithm accordingly.

If you recognize such a situation you can force the use of the sequential blocksplit algorithm by including code such as this in the server:

```
CONTROL TABLE SALES.CUSTOMER SEQUENTIAL BLOCKSPLIT ON;
```

```
    ...  
INSERT INTO SALES.CUSTOMER ... ;      <--- Series of inserts
```

```
    ...  
CONTROL TABLE SALES.CUSTOMER SEQUENTIAL BLOCKSPLIT ENABLE;
```

Reset the SEQUENTIAL BLOCKSPLIT option to ENABLE (as in the final directive) immediately after the sequential inserts. Using the sequential blocksplit algorithm when inserts are not actually sequential can be extremely wasteful of disk space and the disk process normally recognizes sequential inserts without forcing.

- Local autonomy means that a query can complete without error even if some objects or nodes that contribute to the query are unavailable. The SKIP UNAVAILABLE PARTITION option provides local autonomy for certain situations by directing SQL to continue processing a query even if partitions required for the access plan of the query are not available.

For example, assume that a table, CUSTOMER, has three partitions, with first keys 1, 100, and 200 in the CUSTNUM column, and these DEFINE values:

```
$VOL1.SALES.CUSTOMER    =CUST1
$VOL2.SALES.CUSTOMER    =CUST2
$VOL3.SALES.CUSTOMER    =CUST3
```

If partition =CUST2 is unavailable at execution time, the query

```
>> SELECT * FROM =CUST1 WHERE CUSTNUM BETWEEN 50 AND 300;
```

normally fails completely. If you specify the SKIP UNAVAILABLE PARTITION option, however, the query:

- Returns rows with key values between 50 and 99.
- Skips partition =CUST2.
- Returns warning 8239 (Partition was skipped.) with the first row of partition =CUST3.
- Returns all rows of =CUST3 with key values up to 300.

The query proceeds successfully even if you specified the name of the unavailable partition in the FROM clause of the statement. (The partition named in the FROM clause must be available at compile time, however.)

- When an operation is buffered, data is transferred between the file system and the disk process, a block at a time instead of a row at a time. Buffering improves the performance of SQL statements by reducing the number of messages exchanged and the amount of data transferred between the file system and the disk process.

These guidelines apply to sequential block buffering operations enabled by the SEQUENTIAL option:

- Sequential INSERT buffering applies to INSERT operations performed in sequential primary key, clustering key, or SYSKEY order.
- Sequential READ buffering applies to explicit or implicit READ operations performed in sequential primary key, clustering key, SYSKEY, or index order. Note that buffering of READ operations can occur implicitly with a SELECT, UPDATE, DELETE, or cursor statement.
- Sequential UPDATE buffering applies to UPDATE WHERE CURRENT operations and other UPDATE operations performed on a set of sequential rows.
- For INSERT and UPDATE operations on audited tables, any errors returned by the disk process in flushing the buffer cause the current transaction to abort. For nonaudited tables, errors returned by the disk process do not abort the transaction, but might leave the table and its indexes inconsistent. In this case SQL reports a possible loss of data by returning file-system error 122.
- After a sequential INSERT or UPDATE operation has begun, any other DML operation on the same table (directly or through a view using the same

underlying table) within the same process flushes the buffer and interrupts the sequential operation.

- For INSERT and UPDATE operations, any errors the disk process encounters while flushing the buffer are returned to the statement that triggers the buffer flush, rather than to the INSERT or UPDATE statement. For more information, see the discussion on page [C-90](#).
- For sequential read operations that use STABLE access, virtual sequential block buffering (VSBB) can reduce concurrency for other applications that need exclusive locks on the rows in a block. (STABLE access normally provides greater concurrency than REPEATABLE access for sequential read operations, but this is not true with VSBB because the disk process does not release locks on any rows within a virtual block until the cursor moves to the next block.)

To disable VSBB for read operations, use this directive:

```
CONTROL TABLE * SEQUENTIAL READ OFF
```

For inserts into a key-sequenced table that uses a SYSKEY column or a timestamp as the primary key, VSBB is the usual method for insert operations. If concurrent applications are inserting into the table, a high percentage of lock waits and timeouts might occur. (For sequential insert operations, the disk process acquires a range protector lock on the row that follows the last row inserted. If the last row inserted is at the end of the file, the range protector lock is placed at the end of the file, preventing other servers from inserting rows at the end of the table or view.)

To disable VSBB for INSERT operations, use this directive:

```
CONTROL TABLE * SEQUENTIAL INSERT OFF
```

- SQL buffers INSERT or UPDATE operations on nonaudited tables only if the SEQUENTIAL option is ON and SYNCDEPTH is 0. (You can specify both these options with CONTROL TABLE.) For INSERT operations, you must also set the TABLELOCK option to ON (again, with CONTROL TABLE), or use a LOCK TABLE statement that specifies IN EXCLUSIVE MODE. For an INSERT operation, specify FOR REPEATABLE ACCESS also.

To ensure data integrity you must use the FREE RESOURCES statement to flush buffers for nonaudited tables before you exit SQLCI or the host program. For host programs, any flush errors are returned to the SQLCA.

- The conditions listed in this table trigger flushing the INSERT/UPDATE buffer or invalidating the READ buffers. If a problem such as a path error or disk full error occurs during the flush, the disk process returns the error.

Condition	Flushes Buffer for INSERTs and UPDATEs	Invalidates Buffer for READs
A DML operation occurs on a table, interrupting a sequential operation that has already begun.	X	X
The buffer is full.	X	N.A.
A cursor is opened or closed on a table or view with a sequential operation running.	X	N.A.
A TMF transaction completes on an audited table.	X	X
A server replies to a requester.	X	X
A FREE RESOURCES statement is executed by the current process.	X	X
The table is closed.	X	X
The current TMF transaction is not the same as the TMF transaction corresponding to the active INSERT or UPDATE operations.	X	X

The buffer is flushed to the disk process, which puts the buffer in cache and writes the data to disk when the table is closed or, for audited tables, when the transaction commits. For nonaudited tables, the buffer is written to disk when the FREE RESOURCES statement executes.

- SQL does not buffer operations if any of the conditions described in the table occur.

Condition	Prevents Buffering for UPDATEs	Prevents Buffering for INSERTs
A table is not a base table (operations for index maintenance are not buffered).	X	X
A table has at least one cursor defined by the same process.	N.A.	X
A relative or entry-sequenced table is indexed.	N.A.	X
An operation on a relative table does not occur at end-of-file.	N.A.	X
An operation on a relative or entry-sequenced table includes the RETURNING clause with a host variable.	N.A.	X
A nonaudited table has one or more indexes, and the operation involves variable length character columns.	X	N.A.

Condition	Prevents Buffering for UPDATES	Prevents Buffering for INSERTS
A table has more than one cursor defined by the same process, or an alternate access path is selected.	X	N.A.
SEQUENTIAL READ OFF is in effect for the table.	X	N.A.
SEQUENTIAL UPDATE OFF is in effect for the table.	X	N.A.
SEQUENTIAL INSERT OFF is in effect for the table.	N.A.	X
An INSERT or UPDATE operation is mixed with other operations instead of occurring in sequence.	X	X
The INSERT or UPDATE operation is not in sequential order.	X	X

- The optimizer often chooses virtual sequential block buffering (VSBB) when compiling a cursor definition. (You can determine whether SQL used VSBB in a specific case by looking at the EXPLAIN output for the cursor.)

During a cursor session, conflicting DML operations can invalidate the cursor's buffering for the table. Each invalidation forces the next FETCH operation to send a message to the disk process to retrieve a new buffer; this can degrade performance substantially.

These operations invalidate the buffer for cursor operations:

- Any INSERT on the same table by the current process
- A stand-alone UPDATE or DELETE on the same table (directly or through a protection view) from within the same process
- An UPDATE WHERE CURRENT or DELETE WHERE CURRENT using a different cursor to access the same table (directly or through a view) from within the same process

For example, a loop containing both a FETCH statement and a stand-alone UPDATE or DELETE statement on the same table would invalidate the cursor's buffer on every loop iteration. You can change your program logic to minimize or eliminate the performance penalty because of conflicts by doing:

- Avoid INSERT operations within a cursor session.
- Use UPDATE WHERE CURRENT or DELETE WHERE CURRENT operations against the current cursor rather than stand-alone UPDATE or DELETE operations.

- Avoid having a process open multiple cursors on a table when any of the cursors is used to update that table. If this is unavoidable, consider using CONTROL TABLE SEQUENTIAL READ OFF.

Examples—CONTROL TABLE

- This example sets SYNCDEPTH to 0 for the nonaudited table DEPT. You might use such a directive before selecting and displaying all rows of the table, for example, but not before a query that changes data.

```
CONTROL TABLE PERSNL.DEPT SYNCDEPTH 0;
```

- This example requests buffering for sequential INSERT operations on table CUSTOMER:

```
CONTROL TABLE SALES.CUSTOMER SEQUENTIAL INSERT ON;
```

- This example forces SQL to choose a specific access plan for a query that accesses a view created:

```
CREATE VIEW EMPDEPT AS
SELECT EMP_NAME, EMP_NO, E.DEPT_NO, DEPT_NAME, DEPT_LOCN
FROM EMPLOYEE E, DEPT D
WHERE E.DEPT_NO = D.DEPT_NO;
```

The CONTROL TABLE directives force a join sequence and access path for table EMPLOYEE and SALARY, leaving SQL to choose the join sequence for table DEPT. Because the query involves only three tables, the CONTROL TABLE directives implicitly force DEPT to have a join sequence of 1.

```
CONTROL TABLE EMPDEPT AS ED BASETABLE E
ACCESS PATH INDEX IEMP
JOIN SEQUENCE 2;
CONTROL TABLE SALARY
ACCESS PATH PRIMARY
JOIN SEQUENCE 3;
SELECT * FROM EMPDEPT ED, SALARY SA
WHERE ED.EMP_NO = SA.EMP_NO
AND ED.DEPT_NAME = "Engineering";
```

The use of the correlation name ED for the view is not necessary but is included in the first CONTROL TABLE directive to demonstrate that a correlation name can be included for views. The qualification of "BASETABLE E" in the first CONTROL TABLE directive is necessary, however; without it, the CONTROL TABLE options would apply to both the EMPLOYEE and DEPT tables of the view EMPDEPT, and an error would occur.

- This example uses CONTROL TABLE to specify locks. The first directive requests a table lock on table CUSTOMER. These statements lock and unlock the nonaudited table DEPT, using CONTROL TABLE directives with the LOCK and

UNLOCK statements to make sure the compiler considers the locking mode when selecting an access path for DEPT.

```
CONTROL TABLE SALES.CUSTOMER TABLELOCK ON;
LOCK TABLE PERSNL.DEPT IN EXCLUSIVE MODE;
CONTROL TABLE PERSNL.DEPT TABLELOCK ON;
...
UNLOCK TABLE PERSNL.DEPT;
CONTROL TABLE PERSNL.DEPT TABLELOCK ENABLE;
```

- This example shows how to force SQL to use a specific index, join order, and join method for a single SELECT statement, then return control over access path, join sequence, and join method for future SELECT operations to SQL. The example uses table EMPLOYEE with a primary key of EMP_NO and an alternate index, IEMP, on column DEPT_NO.

```
CONTROL TABLE EMPLOYEE AS E1 ACCESS PATH INDEX IEMP
                                JOIN SEQUENCE 1;
CONTROL TABLE EMPLOYEE AS E2 ACCESS PATH PRIMARY
                                JOIN SEQUENCE 2
                                JOIN METHOD NESTED;
SELECT E1.EMP_NAME, E1.DEPT_NAME, E2.EMP_NAME, E2.DEPT_NAME
FROM EMPLOYEE E1, EMPLOYEE E2
WHERE E1.DEPT_NO = "9999" AND E1.MGR_ENO = E2.EMP_NO;
CONTROL TABLE * ACCESS PATH SYSTEM
                                JOIN SEQUENCE SYSTEM
                                JOIN METHOD SYSTEM;
```

The CONTROL TABLE directives that precede the SELECT force SQL to use two different access paths to access two different instances of table EMPLOYEE (IEMP for E1, and primary access for E2). The statements also force SQL to use a specified join order for processing the data, and to use a nested join method to join the second table (E2) with the first (E1).

- This example forces MDAM with 3 key columns for table T1:

```
CONTROL TABLE T1 ACCESS PATH PRIMARY MDAM ON USE 3 COLUMNS;
```

- This example cancels all previously set CONTROL TABLE options and uses only the default values for the options:

```
CONTROL TABLE *;
```

CONVERT Command

CONVERT is an SQLCI utility that creates an EDIT file containing SQL commands to convert an Enscribe file described in a Data Definition Language (DDL) dictionary to an SQL table described in a specific catalog. You can invoke the EDIT file commands by using the OBEY command to create:

- a table containing rows that correspond to the record definition of the Enscribe file and columns that correspond to the fields in each record.
- indexes on the table that correspond to the alternate keys associated with the Enscribe file.

If SMF is installed on your node, files you specify in CONVERT syntax cannot be on any `$.ZYS*` subvolumes.

```

CONVERT RECORD ddl-record-name TO TABLE table-name

  [ [,] convert-option ] ... ;

convert-option is:

{ MAP NAME[S] { map
{               { ( map [ , map ] ... ) }
{               }
{ CATALOG[S] { catalog-spec
{              { ( catalog-spec [, catalog-spec ]...)
{              }
{ COMMENTS
{
{ DICTIONARY subvol
{
{ FILE IS enscribe-file
{
{ { LOAD | NO LOAD }
{
{ { PART | NO PART }
{
{ SOURCE edit-file [ CLEAR ]
{
{ { VARCHARS | NO VARCHARS }
{
{ REDEFINE ( redef-spec [, redef-spec ] ... )
{

```

```
{ CHARACTER { ISO88591 }
{             { ISO88592 }
{             { ISO88593 }
{             { ISO88594 }
{             { ISO88595 }
{             { ISO88596 }
{             { ISO88597 }
{             { ISO88598 }
{             { ISO88599 }
{             { KANJI }
{             { KSC5601 }
{             { UNKNOWN }
{
{ NATIONAL { ISO88591 }
{           { ISO88592 }
{           { ISO88593 }
{           { ISO88594 }
{           { ISO88595 }
{           { ISO88596 }
{           { ISO88597 }
{           { ISO88598 }
{           { ISO88599 }
{           { KANJI }
{           { KSC5601 }
{           { UNKNOWN }
{           { DEFAULT }
{ }
```

map is:

simple-fileset-list TO files

catalog-spec is:

catalog-name [FOR simple-fileset-list]

redef-spec is:

original-qualified-name AS redefined-qualified-name

ddl-record-name

is a DDL data name that identifies the DDL record definition for the file to convert. The record definition must be in the DDL dictionary specified by the DICTONARY clause (or, if the DICTONARY clause is omitted, the DDL dictionary on the default subvolume).

table-name

specifies a Guardian name (or an equivalent DEFINE) for the new table.

```
MAP NAME[S] { map
              { ( map [ ,map ] ... ) } }
```

overrides the default names (and locations) used to generate names for secondary partitions of tables and for indexes.

map is:

```
simple-fileset-list TO files
simple-fileset-list
```

is a simple fileset list that specifies the secondary partitions and alternate-key files of the Enscribe file to map. For more information about simple fileset lists, see [Filesets](#) on page F-29.

TO *files*

specifies the names and locations for the new partitions and indexes.

files is a Guardian name that can optionally contain an asterisk to indicate that the portion of the name in which the asterisk appears should be the same as the name of the corresponding portion of the source element in *simple-fileset-list*. For example,

```
$VOL1.SUBV2.* TO *.PERSNL.*
```

specifies that the partitions or indexes will be on the volume and subvolume \$VOL1.PERSNL and will have the same names as the source objects.

If you specify more than one *map* and one conflicts with another, CONVERT uses the first map.

If you do not specify the MAP NAMES option, CONVERT produces a command to create a secondary partition on the same volume and subvolume as its corresponding Enscribe file partition. CONVERT produces a command to create indexes on the same volume and subvolume as the new table.

```
CATALOG[S] { catalog-spec
             { ( catalog-spec [ , catalog-spec ]...) } }
```

specifies the catalogs in which to describe the target objects. *catalog-spec* is:

```
catalog-name [ FOR simple-fileset-list ]
```

catalog-name identifies the catalog to hold the descriptions of the objects.

FOR *simple-fileset-list* specifies which target objects to describe in the catalog. Use the names of the converted objects, not the source objects or files. If you omit this clause, the catalog is used for the descriptions of all target objects.

If you omit the CATALOGS option, SQL describes the new table and indexes in the current default catalog.

COMMENTS

writes all qualifier names and their corresponding level numbers as comments in the EDIT file specified in the SOURCE option (or the default EDIT file, CNVSRC).

DICTIONARY *subvol*

specifies the name of the subvolume (or an equivalent DEFINE) that contains the DDL record definition *ddl-record-name*.

The default is the current default subvolume.

FILE IS *enscribe-file*

specifies the name (or an equivalent DEFINE) of the Enscribe file to convert.

The default is the file name in the FILE IS clause of the DDL record definition name that follows the keyword RECORD.

{ LOAD | NO LOAD }

specifies or inhibits loading data from the file to be converted into the table. If you specify NO LOAD, you can use the LOAD command later to load the data.

The default is LOAD.

{ PART | NO PART }

specifies whether to convert a partitioned Enscribe file to a partitioned table or a nonpartitioned table.

The default is PART.

SOURCE *edit-file* [CLEAR]

identifies an EDIT file to contain the SQL commands. You can examine these commands and modify them if necessary before executing the file with OBEY.

edit-file is the name of the file. CONVERT creates the file if it does not exist.

CLEAR clears all data from the file before adding the command. If you omit CLEAR, CONVERT appends commands to the end of the file.

The default *edit-file* is CNVSRC.

{ VARCHARS | NO VARCHARS }

specifies or inhibits conversion of the special DDL group that represents varying-length strings to the SQL VARCHAR data type:

VARCHARS	converts the two elementary fields to one column with data type VARCHAR
NO VARCHARS	converts the two elementary fields to one column with data type CHAR

The default is VARCHARS.

REDEFINE (*redef-spec* [, *redef-spec*] ...)

specifies that original items (groups or fields) are to be converted to columns based on redefinitions of the items. Unless you include the REDEFINE option, all items are converted according to the definition of the original items, and the REDEFINES clause in the DDL record definition is ignored.

redef-spec is:

original-qualified-name AS *redefined-qualified-name*
original-qualified-name

identifies an original field or group in a DDL record. The name must be qualified by the group names at all preceding levels; for example, CUSTOMER.ADDRESS.STREET-ADDRESS is the qualified name for the STREET-ADDRESS field of the ADDRESS group. The ADDRESS group is in the CUSTOMER group.

redefined-qualified-name

identifies a redefined field or group that corresponds to the original field or group. The name must be qualified by the group names at all preceding levels; for example, CUSTOMER.ADDRESS.STREET-DETAIL is the qualified name of the STREET-DETAIL field that redefines the STREET-ADDRESS field.

When the source field is converted, the conversion is based on the redefinition that you specify.

If the item is redefined as a group, the elementary fields of the group are converted to columns. When the redefined item is shorter than the original item, the resulting columns are the size of the redefined item; CONVERT does not pad the columns with blanks.

CHARACTER { ISO88591 | ISO88592 | ... | UNKNOWN }

specifies the character set for PIC X, PIC A, and TYPE CHARACTER fields. If you omit the CHARACTER option, all items are converted according to the definitions of the original items.

The character set can be one of the single-byte character sets ISO 8859/1 through ISO 8859/9, or one of the double-byte character sets Kanji or KSC5601. (For more information, see [Character Sets](#) on page C-17.)

UNKNOWN specifies that the character set is unknown, and specifying this option is equivalent to omitting the CHARACTER clause. SQL uses the data as 8-bit data.

For more information about how the conversion is performed, see [Conversion of DDL Elementary Items](#) on page C-103.

`NATIONAL { ISO88591 | ISO88592 | ... | DEFAULT }`

specifies the character set for PIC N fields. If you omit the NATIONAL option, all items are converted according to the definitions of the original items.

The character set can be one of the single-byte character sets ISO 8859/1 through ISO 8859/9, or one of the double-byte character sets Kanji or KSC5601. (For more information, see [Character Sets](#) on page C-17.)

UNKNOWN specifies that the character set is unknown, and specifying this option is equivalent to omitting the CHARACTER clause. SQL uses the data as 8-bit data.

DEFAULT specifies the system default multibyte character set.

For more information about how the conversion is performed, see [Conversion of DDL Elementary Items](#) on page C-103.

CONVERT Behavior

CONVERT writes these to the EDIT file:

- A section header in the form `?SECTION CREATE_`*table-name*
- A CREATE TABLE command that defines a table compatible with the DDL record definition for the Enscribe file
- A CREATE INDEX command for each alternate-key file associated with the Enscribe file
- A section header in the form `?SECTION LOAD_`*table-name*
- A LOAD command (which includes the REDEFINE option you specify in the CONVERT command) that loads data from the Enscribe file into the empty table and data from the alternate-key files associated with the Enscribe file into the empty indexes

CONVERT resolves any DEFINE names you specify in the CONVERT command and uses the actual file names in the commands it writes. Except for table name in ?SECTION headers, all file names are fully qualified. You can examine and modify the text of the file if necessary. You use OBEY to execute the commands in the file.

Because a load operation cannot run within a TMF transaction, you cannot execute the OBEY command generated by CONVERT within a user-defined transaction. The catalog manager defines transactions while the CREATE commands are executing. For more information about how audited tables are loaded, see [LOAD Command](#) on page L-18.

CONVERT requires authority to read the DDL dictionary and authority to write to the EDIT file that receives the generated commands.

Executing the EDIT file with OBEY requires authority to read the EDIT file; authority to write to the affected catalogs (for creating the table and indexes); authority to read the DDL dictionary, the Enscribe source file, and the catalog in which the table is described; and authority to read and write to the table (for loading the table and indexes).

If you press the Break key while the CONVERT command is executing and the BREAK_KEY option is ON, SQLCI stops execution, displays a message stating that the command was terminated by a Break, and displays the standard SQLCI prompt. The EDIT file is closed. If the BREAK_KEY option is OFF, control returns to the previous Break key owner (usually the command interpreter process).

The CONVERT utility does not check any version information; therefore, the versions of the table and any indexes you are creating cannot be greater than the version of the catalog in which they will be registered.

CONVERT supports the HEADING, UPSHIFT, and HELP TEXT attributes.

Enscribe Files and DDL Record Definitions

CONVERT operates on Enscribe files only. CONVERT derives a table's file organization and primary key location from the Enscribe file and derives column names and column data types from the DDL record definition for the Enscribe file. CONVERT also generates commands to create indexes on the table from the alternate key specifications in the Enscribe file.

When a structured Enscribe file is converted to a table, the table is assigned the same file organization as the file: key-sequenced, relative, or entry-sequenced. An unstructured file is converted to an entry-sequenced table.

The file must have an associated DDL record definition. The actual file organization and record length must be the same as the specifications for these attributes in the DDL record definition.

If a DDL record definition defines an elementary item or group by means of a DDL DEF structure, CONVERT uses the DEFs to convert the record definition, but a DEF does not translate to an SQL data structure. During the conversion of a record definition, CONVERT echoes the SQL table definition on the home terminal screen. (If you work with data that includes multibyte characters, be aware that column DEFAULT clauses can contain multibyte characters unsupported by the terminal and this can cause unpredictable results in the screen display.)

DDL Primary Keys and Alternate Keys

CONVERT uses the primary key specification in the Enscribe file to assign the primary key of the table. The SQL primary key is defined on columns derived from a field or fields specified in the DDL record definition. If the key definition spans multiple fields, a multicolumn SQL primary key is produced.

CONVERT uses the alternate key specifications in the Enscribe file to define indexes on the table. Because SQL requires a separate index file for each alternate key, a CREATE INDEX command is generated for each alternate key named in the DDL record definition.

Indexes are created on the same volume and subvolume as the table. Index names are created by appending numbers to the table name. The name of the tables, therefore, should be shorter than eight characters. For example, the first index of the DEPT table is DEPT0, the second is DEPT1, and so forth.

You can use the MAP NAME option to override this naming pattern. Using the MAP NAMES option, you can map the name of an alternate-key file to an index name. If other index names are needed, they are created by appending numbers to the index name specified in the MAP NAMES option.

The key specifier for each alternate key in the DDL record definition is used in the KEYTAG clause of the CREATE INDEX command to associate a key specifier with an index.

If a field name defined with a DDL REDEFINES clause is used as a primary or alternate key, CONVERT determines which column is to be the key based on the REDEFINE option of the CONVERT command. If you do not include the REDEFINE option, CONVERT uses the original field as the key column.

For example, if the primary key CURRENT_DATA redefines EMPINFO, CONVERT uses EMPINFO—not CURRENT_DATA—as the primary key. You can use the REDEFINE option to specify that you want to use the redefined field instead of the original one. For example, if you specify REDEFINE (EMPINFO AS CURRENT_DATA), the CURRENT_DATA is used as the primary key.

Indexes are automatically loaded with data when the table is loaded.

DDL Clause Mapping

This DDL record definition clauses define constructs that are not used in SQL:

- OCCURS and OCCURS DEPENDING ON clauses define repeating fixed or varying groups or arrays. For SQL, each elementary item in the OCCURS group is converted to a table column, and the table column is repeated a fixed number of times. The names of repeated columns are derived from the DDL elementary item name combined with the array index value. For OCCURS MIN TO MAX TIMES DEPENDING ON *field-name*, table columns are repeated MAX number of times.
- Fields with FILLER clauses are skipped.
- Conversion of a VALUE IS clause depends on the CONVERT option specified. The two forms of the DDL VALUE IS clause used with PIC X, PIC A, and TYPE CHARACTER fields are:

```
VALUE IS "default-string"
VALUE IS ALL "default-character"
```

The two forms of the DDL VALUE IS clause used with PIC N fields are:

```
VALUE IS N"default-string"
VALUE IS ALL N"default-character"
```

If the CHARACTER option, the NATIONAL option, or the NATIONAL DEFAULT option is not specified, the VALUE IS clause is converted:

```
DEFAULT "default-string"
DEFAULT "default-character-repeated"
default-string
```

specifies a default value for the column. If the *default-string* value specified in the DDL VALUE IS clause is longer than eight bytes, the value is truncated to eight bytes and a warning is displayed.

```
default-character-repeated
```

specifies either a single-byte character (for PIC X, PIC A, or TYPE CHARACTER fields) or a double-byte character (for PIC N fields) that is repeated to make a total of eight bytes. If more than one character is specified, only the first character is repeated.

- If the CHARACTER option or the NATIONAL option is specified, the VALUE IS clauses are converted:

```
DEFAULT _char-set-name "default-string"
DEFAULT _char-set-name "default-character-repeated"
character-set-name
```

specifies the character set designated by either the CHARACTER or NATIONAL option in the CONVERT command.

- If the NATIONAL DEFAULT option is specified, the PIC N VALUE IS clauses are converted:

```
DEFAULT N"default-string"
DEFAULT N"default-character-repeated"
```

- CONVERT ignores these clauses:
 - Level-66 RENAMES clauses
 - Level-88 CONDITION-NAME clauses
 - DISPLAY clauses
 - MUST BE clauses

Conversion of DDL Elementary Items

CONVERT converts each elementary field item in the record definition to a column definition. The table that is created as a result of the conversion can contain at most 450 columns.

CONVERT uses the name of a DDL field as the name for the corresponding column but replaces any hyphens (-) in the name with underscores (_).

If the data type of a DDL field is equivalent to an SQL data type, the column is assigned the equivalent data type. If the data type is not equivalent to any SQL data type, the column is assigned a compatible data type, if possible. If there is no compatible data type, the column specification becomes a comment, and a warning is appended.

DDL fields of various data types are converted as shown in the table.

Conversion of DDL Character Strings

Option	DDL Data Type	SQL Data Type
None	PIC A(<i>j</i>)	PIC X(<i>j</i>)
	PIC X(<i>j</i>)	PIC X(<i>j</i>)
	PIC N(<i>k</i>)	PIC X(<i>m</i>)
	TYPE CHARACTER <i>j</i>	CHAR(<i>j</i>)
CHARACTER <i>set</i>	PIC A(<i>j</i>)	CHAR(<i>x</i>) CHARACTER SET <i>set</i>
	PIC X(<i>j</i>)	CHAR(<i>x</i>) CHARACTER SET <i>set</i>
	TYPE CHARACTER <i>j</i>	CHAR(<i>x</i>) CHARACTER SET <i>set</i>
NATIONAL <i>set</i>	PIC N(<i>k</i>)	CHAR(<i>y</i>)
		CHARACTER SET <i>set</i>
NATIONAL DEFAULT	PIC N(<i>k</i>)	NCHAR(<i>k</i>)

set An SQL-supported character set

j The number of single-byte characters

k The number of double-byte characters

m The number of characters in the SQL column; *m* is two times the corresponding *k*

x The number of characters; *x* equals *j* for single-byte characters sets and equals *j*/2 for double-byte character sets

y The number of characters; *y* equals *k* for double-byte character sets and equals twice *k* for single-byte character sets

Conversion of Binary Data TypesConversion of Decimal Data

DDL	NonStop SQL/MP
PIC <i>nines</i> COMP	PIC <i>nines</i> COMP or if precision is greater than 9, PIC S <i>nines</i> COMP
PIC S <i>nines</i> COMP	PIC S <i>nines</i> COMP
TYPE BINARY 8	PIC X(1)
TYPE BINARY 8 UNSIGNED	PIC X(1)
TYPE BINARY 16	SMALLINT
TYPE BINARY 16 UNSIGNED	SMALLINT UNSIGNED
TYPE BINARY 16, <i>scale</i>	NUMERIC (4, <i>scale</i>)
TYPE BINARY 16, <i>scale</i> UNSIGNED	NUMERIC (4, <i>scale</i>) UNSIGNED
TYPE BINARY 32	INTEGER
TYPE BINARY 32 UNSIGNED	INTEGER UNSIGNED
<i>nines</i> is:	$\left\{ \begin{array}{l} 9 \text{ (int [V9 (scale)])} \\ \text{V9 (scale)} \end{array} \right\}$

DDL	NonStop SQL/MP
TYPE BINARY 32, <i>scale</i>	NUMERIC (9, <i>scale</i>)
TYPE BINARY 32, <i>scale</i> UNSIGNED	NUMERIC (9, <i>scale</i>) UNSIGNED
TYPE BINARY 64	LARGEINT
TYPE BINARY 64, <i>scale</i>	NUMERIC (18, <i>scale</i>)
<i>nines</i> is: { 9 (<i>int</i> [V9 (<i>scale</i>)] } { V9 (<i>scale</i>) }	

Types

DDL	NonStop SQL/MP
PIC <i>nines</i>	PIC <i>nines</i> or if precision is greater than 9, PIC S <i>nines</i>
PIC S <i>nines</i>	PIC S <i>nines</i>
PIC <i>nines</i> S	PIC S <i>nines</i>
PIC T	PIC S9
PIC T9(<i>int</i>)	PIC S9(<i>int</i> +1)
PIC TV9(<i>scale</i>)	PIC S9V9(<i>scale</i>)
PIC T9(<i>int</i>) V9(<i>scale</i>)	PIC S9(<i>int</i> +1)V9(<i>scale</i>)
PIC 9(<i>int</i>)T	PIC S9(<i>int</i> +1)
PIC V9(<i>scale</i>)T	PIC SV9(<i>scale</i> +1)
PIC 9(<i>int</i>) V9(<i>scale</i>)T	PIC S9(<i>int</i>)V9(<i>scale</i> +1)
<i>nines</i> is: { 9 (<i>int</i> [V9 (<i>scale</i>)] } { V9 (<i>scale</i>) }	

Note. SQL supports only the unsigned and signed LEADING EMBEDDED DECIMAL data types. The DDL SIGN LEADING SEPARATE or TRAILING SEPARATE DECIMAL data types are converted to the SQL SIGN LEADING EMBEDDED DECIMAL data type.

Conversion of FORTRAN Data Types

DDL	NonStop SQL/MP
TYPE COMPLEX	PIC X(8)
TYPE FLOAT 32	REAL
TYPE FLOAT 64	DOUBLE PRECISION

DDL	NonStop SQL/MP
TYPE LOGICAL 1	PIC X
TYPE LOGICAL 2	SMALLINT
TYPE LOGICAL 4	INTEGER

This special DDL group is converted to the SQL VARCHAR data type:

```
02 A-VARCHAR.
   03 LEN PIC S9(4) COMP.
   03 VAL PIC N(len).
```

The field names in the special DDL group have these meanings:

- A-VARCHAR is the name of the special DDL VARCHAR group. Any valid DDL group name can be used in place of A-VARCHAR.
- LEN is a numeric field representing the actual length of the string. The string must be defined as LEN PIC S9(4) COMP for the group to be recognized as this special DDL group.
- VAL is a fixed-length character field representing the maximum length of the string. The data type of the character field can be PIC X(*n*), PIC A(*n*), TYPE CHARACTER, or PIC N(*n*).

This table shows how variable-length character strings are converted, depending on the option you specify.

Conversion of Variable-Length Strings

With VARCHARS Option

CONVERT Option	SQL Data Type PIC N Field	SQL Data Type PIC X, PIC A or CHARACTER Field
None	VARCHAR(<i>m</i>)	VARCHAR(<i>j</i>)
CHARACTER <i>set</i>		VARCHAR(<i>x</i>) CHARACTER SET <i>set</i>
NATIONAL <i>set</i>	VARCHAR(<i>y</i>) CHARACTER SET <i>set</i>	
NATIONAL DEFAULT	NCHAR VARYING(<i>k</i>)	

set An SQL-supported character set

j The number of single-byte characters

k The number of double-byte characters

m The number of characters; *m* is twice the corresponding *k*

x The number of characters; *x* equals *j* for single-byte character sets and equals *j*/2 for double-byte character sets

y The number of characters; *y* equals *k* for double-byte character sets and equals twice *k* for single-byte character sets

With VARCHARS Option

CONVERT Option	SQL Data Type PIC N Field	SQL Data Type PIC X, PIC A or CHARACTER Field
Without VARCHARS Option		
None	CHAR(<i>m</i>)	CHAR(<i>j</i>)
CHARACTER <i>set</i>		CHAR(<i>x</i>)
		CHARACTER SET <i>set</i>
NATIONAL <i>set</i>	CHAR(<i>y</i>)	
	CHARACTER SET <i>set</i>	
NATIONAL DEFAULT	NCHAR(<i>k</i>)	

set An SQL-supported character set

j The number of single-byte characters

k The number of double-byte characters

m The number of characters; *m* is twice the corresponding *k*

x The number of characters; *x* equals *j* for single-byte character sets and equals *j*/2 for double-byte character sets

y The number of characters; *y* equals *k* for double-byte character sets and equals twice *k* for single-byte character sets

DDL Groups

SQL does not let you define groups of columns. When the record definition is converted to a table definition, each elementary item in a group is translated to a corresponding column. The group name is not translated.

If two fields in the same record definition convert to the same column name, a different digit is appended to each one to make the names unique.

Physical File Attributes of Tables and Indexes

CONVERT derives most of the physical file attributes of a table directly from the physical attributes of the Enscribe file being converted. For indexes, CONVERT derives MAXEXTENTS and EXTENT file attributes from the physical attributes of the Enscribe alternate key files. Attributes that CONVERT does not specifically set in the CREATE INDEX statement default to predetermined values.

For the AUDIT attribute, the file is created with the same value as the original file if you specify NO LOAD; otherwise, the file is created with NO AUDIT, then set to AUDIT after loading if the original file was audited.

Partition Attributes of Tables and Indexes

If you convert a partitioned Enscribe file to a partitioned table, these rules apply:

- The partitions of the table are derived from the partitions of the file. Each table partition, other than the first one, is created on the same subvolume as the corresponding file partition, unless you use the MAP NAMES option to override this naming pattern.
- The EXTENT and MAXEXTENTS physical attributes for each table partition are derived from the corresponding Enscribe file partition.
- For key-sequenced files, each Enscribe file partition's LOW KEY attribute is used to derive the FIRST KEY attribute of the corresponding table partition. An Enscribe LOW KEY value must be a valid FIRST KEY value in a CREATE TABLE command.
- Partitioned Enscribe alternate key files are converted to partitioned SQL indexes.

Examples—CONVERT

- Suppose that you have a key-sequenced Enscribe file named ORDERTAB on subvolume \$VOL3.DDL. The contents of the Enscribe file are described by the record definition ORDER. The record ORDER includes two DDL groups named ORDERDATE and DELDATE. The record definition also indicates that the primary key is ORDERNUM and that the alternate key field is CUSTNUM.

Here is the DDL record definition for the ORDER file:

```
RECORD ORDER.
FILE IS "$VOL3.DDL.ORDERTAB"  KEY-SEQUENCED.
02 ORDERNUM          PIC 9(3).
02 ORDERDATE.
    03 MONTH          PIC 9(2).
    03 DAY            PIC 9(2).
    03 YEAR           PIC 9(2).
02 DELDATE.
    03 MONTH          PIC 9(2).
    03 DAY            PIC 9(2).
    03 YEAR           PIC 9(2).
02 SALES-PERSON      PIC X(4).
02 BRANCHNUM         TYPE BINARY 32 NULL RaS VALUE IS 9999.
02 CUSTNAME.
    03 LEN            PIC S9(4) COMP.
    03 VAL            PIC X(30) VALUE IS "RINTERNALS".
02 CUSTNUM           PIC 9(4).
02 STATUS            PIC X(8) VALUE IS "RON HOLDS".
02 TOTAL-AMOUNT      PIC 9(5)V9(2) NULL 0.
KEY IS ORDERNUM DUPPLICATES NOT ALLOWED.
KEY "OC" IS CUSTNUM.
END
```

- Suppose that your current default subvolume is \$VOL3.DDL and that you want to convert the Enscribe file ORDERTAB to a table named ORDERS on the

subvolume \$VOL1.SALES. ORDERTAB resides on the current default subvolume and is described by the record definition ORDER in a DDL dictionary. The SQL catalog in which you want ORDERS to be described also resides on the subvolume \$VOL1.SALES. This command creates an EDIT file named CNVSRC that contains the commands needed to perform the conversion:

```
>> CONVERT RECORD ORDER TO TABLE $VOL1.SALES.ORDERS
```

```
>+ CATALOG $VOL1.SALES;
```

CNVSRC contains these commands:

```
?SECTION CREATE_ORDERS
CREATE TABLE \NODE.$VOL1.SALES.ORDERS
( ORDERNUM          PIC 9(3)  NOT NULL,
  MONTH             PIC 9(2)  NOT NULL,
  DAY               PIC 9(2)  NOT NULL,
  YEAR              PIC 9(2)  NOT NULL,
  MONTH2            PIC 9(2)  NOT NULL,
  DAY2              PIC 9(2)  NOT NULL,
  YEAR2             PIC 9(2)  NOT NULL,
  SALES_PERSON      PIC X(4)  NOT NULL,
  BRANCHNUM         INTEGER
                      DEFAULT 9999,
  CUSTNAME          VARCHAR(30)
                      DEFAULT "INTERNAL" NOT NULL,
  CUSTNUM           PIC 9(4)  NOT NULL,
  STATUS            PIC X(8)
                      DEFAULT "ON HOLD" NOT NULL,
  TOTAL_AMOUNT     PIC 9(5)V9(2),
  PRIMARY KEY ORDERNUM
)
  ORGANIZATION KEY SEQUENCED
  CATALOG \NODE.$VOL1.SALES
  BLOCKSIZE 4096
  EXTENT (4,32)
  MAXEXTENTS 100
  TABLECODE 0
  NO AUDIT
  NO CLEARONPURGE
  NO DCOMPRESS
  NO ICOMPRESS
  NO SERIALWRITES
  NO VERIFIEDWRITES
  NO BUFFERED;
CREATE INDEX \NODE.$VOL1.SALES.ORDERS0
  ON \NODE.$VOL1.SALES.ORDERS (CUSTNUM)
  CATALOG \SYSTEM.$VOL1.SALES
  KEYTAG "oc"
  EXTENT (4,32)
  MAXEXTENTS 100;
?SECTION LOAD_ORDERS
LOAD \NODE.$VOL3.DDL.ORDERTAB, \SYSTEM.$VOL1.SALES.ORDERS,
SOURCEDICT \NODE.$VOL3.DDL,
```

```
SOURCE REC ORDER,
USERSQLNULLS;
```

Note that the record definition has been altered by eliminating the DDL group names and by creating a column for each elementary item in each group. CONVERT makes these column names unique by appending 2 to the MONTH, DAY, and YEAR columns derived from the DDL DELDATE group. You can change these names by editing the file.

This command performs the conversion:

```
>> OBEY CNVSRC;
```

- This example is the DDL record definition for the key-sequenced Enscribe file called TSTKANJI and uses the Kanji data type:

```
RECORD kanji.
FILE IS "$voll1.subvoll1.tstkanji" KEY-SEQUENCED.
02 A PIC N.
02 B PIC N VALUE IS N"aa".
02 C PIC N(4).
02 D PIC N(3) VALUE IS N"abcdef".
02 E PIC N(5) VALUE IS N"abcdefghij".
02 F PIC N(4) REDEFINES E.
02 G.
    03 LEN PIC S9(4) COMP.
    03 VAL PIC N(4).
KEY IS C DUPLICATES NOT ALLOWED.
END
```

- This example creates an EDIT file named CNVSRC that contains the commands needed to convert the DDL record definition to an SQL table:

```
>> CONVERT RECORD kanji TO TABLE $vol2.subvol2.kanjitbl;
```

Note that the CONVERT command does not include the REDEFINE clause, so F in the DDL record definition is ignored and does not redefine E.

CNVSRC contains this CREATE TABLE command:

```
?SECTION CREATE_KANJITBL
CREATE TABLE \NODE.$VOL2.SUBVOL2.KANJITBL
(
A          PIC X(2) NOT NULL ,  -- WARNING - PIC N
B          PIC X(2)  -- WARNING - PIC N
          DEFAULT "aa" NOT NULL ,
--WARNING Default literal originally national language
string
C          PIC X(8) NOT NULL ,  -- WARNING - PIC N
D          PIC X(6)  -- WARNING - PIC N
          DEFAULT "abcdef" NOT NULL ,
--WARNING Default literal originally national language
string
E          PIC X(10)  -- WARNING - PIC N
          DEFAULT "abcdefgh" NOT NULL ,
--WARNING Default literal originally national language
```

```

string
--WARNING  The default value is truncated.
G          VARCHAR(8) NOT NULL ,  -- WARNING - PIC N
PRIMARY KEY C
)

```

If the DDL default string is longer than eight characters, CONVERT truncates the default string to eight bytes long.

- This example illustrates more features of the CONVERT command. The DDL record definition follows:

```

RECORD SCHEDULE.
FILE IS "$VOL3.DDL.SCHEDULE"  KEY-SEQUENCED.
  02 EMP-SCHEDULE.
    03 EMPNUM                  PIC 9(5).
    03 EMPNUM-KEY              REDEFINES EMPNUM.
      05 DEP-KEY               PIC X(2).
      05 EMP-KEY               PIC X(3).
    03 DAY-SCHED OCCURS 5 TIMES.
      04 DAYNUM                PIC X(2).
      04 SHIFTS OCCURS 2 TIMES.
        05 START-HOUR          PIC 9(2).
        05 END-HOUR            PIC 9(2).
KEY IS EMPNUM  DUPLICATES NOT ALLOWED.
END

```

- This command produces an EDIT file named SCHDCONV, which contains commands that convert the Enscribe file to a table and includes comments in the CREATE TABLE command:

```

>>CONVERT RECORD SCHEDULE TO TABLE $VOL1.PERSNL.SCHEDULE
+>REDEFINE(EMP-SCHEDULE.EMPNUM AS EMP-SCHEDULE.EMPNUM-KEY)
+>COMMENTS SOURCE SCHDCONV CLEAR;

```

The EMPNUM field is converted to two columns named DEP_KEY and EMP_KEY based on the elementary fields of the EMPNUM-KEY redefinition. The column definitions in the CREATE TABLE command are:

```

DEP_KEY      PIC X(2),
EMP_KEY      PIC X(3),

```

The LOAD command in the SCHDCONV file contains the REDEFINE (EMP-SCHEDULE.EMPNUM AS EMP-SCHEDULE.EMPNUM-KEY) option. The CREATE TABLE command follows:

```

CREATE TABLE $VOL1.PERSNL.SCHEDULE
(
  -- 02  EMP-SCHEDULE
  -- 03  EMPNUM-KEY
  -- 04  DEP-KEY
DEP_KEY                      PIC X(2) NOT NULL,
  -- 04  EMP-KEY
EMP_KEY                      PIC X(3) NOT NULL,

```

```

--      03      DAY-SCHED  OCCURS 1/5 TIMES
--      04      DAYNUM
DAYNUM_1                                     PIC X(2) NOT NULL,
--      04      SHIFTS OCCURS 1/2 TIMES
--      05      START-HOUR
START_HOUR_1_1                             PIC 9(2) NOT NULL,
--      05      END-HOUR
END_HOUR_1_1                               PIC 9(2) NOT NULL,
--      05      START-HOUR
START_HOUR_1_2                             PIC 9(2) NOT NULL,
--      05      END-HOUR
END_HOUR_1_2                               PIC 9(2) NOT NULL,
...
DAYNUM_5                                     PIC X(2) NOT NULL,
--      04      SHIFTS OCCURS 1/2 TIMES
--      05      START-HOUR
START_HOUR_5_1                             PIC 9(2) NOT NULL,
...

END_HOUR_5_2                               PIC 9(2) NOT NULL,
PRIMARY KEY (
    DEP_KEY ,
    EMP_KEY
)
)
ORGANIZATION KEY SEQUENCED
...
NO BUFFERED
;

```

Each elementary item of the OCCURS group is converted to a column. The two fields of the EMPNUM-KEY group, which redefined EMPNUM, are used as the primary key of the table. This command performs the conversion:

```
>> OBEY SCHDCONV;
```

CONVERTTIMESTAMP Function

CONVERTTIMESTAMP is a function that converts a Julian timestamp to a DATETIME value. It returns a value of DATETIME that has the range of fields YEAR TO FRACTION(6).

```
CONVERTTIMESTAMP ( julian-timestamp )
```

julian-timestamp

is an expression that evaluates to a Julian timestamp, which is a LARGEINT value.

Example—CONVERTTIMESTAMP

This example converts a Julian timestamp into a DATETIME value:

```
>> SELECT CONVERTTIMESTAMP (HIRE_DATE) FROM EMPLOYEE;
```

COPY Command

COPY is an SQLCI utility command that copies data to and from Guardian files (including Guardian processes and devices, unstructured disk files, and Enscribe structured disk files) and SQL tables, appending the data to any existing data, or displays the contents of a file or table. If you copy data to a table, COPY automatically updates the indexes of the table.

COPY resembles the FUP COPY command but, unlike FUP COPY, COPY works with SQL objects.

If SMF is installed on your node, files you specify in COPY syntax cannot be on any \$*.ZYS*. subvolumes. However, remote files on a non-SMF node can reside on any subvolume.

```
COPY in-file [ [, out-file [ [,] option ] ... ] ] ;
           [ ,, option [ [,] option ] ... ]
```

option is:

```
{ control-option }
{ in-option       }
{ out-option      }
{ display-option }
{ move-option     }
```

control-option is:

```
{ ALLOWERRORS [ ON | OFF | num ] }
{ COUNT num-records }
{ FIRST { ordinal-record-num } }
{       { KEY record-spec } }
{       { KEY (key-value [, key-value]... ) } }
{       { key-specifier ALTKEY (key-value } }
{       { [, key-value ] ... ) } }
{ REPLACE SPACES WITH { ZERO[ES] | DEFAULT[S] } }
{ UNSTRUCTURED }
{ UPSHIFT }
{ USESQLNULLS }
{ SQLNULLABLE }
```

in-option is:

```
{ BLOCKIN in-block-length }
{ { COMPACT | NO COMPACT } }
{ EBCDICIN }
{ RECIN in-record-length }
{ REELS num-reels }
{ { REWINDIN | NO REWINDIN } }
{ SHARE }
{ SKIPIN num-eofs }
{ TRIM trim-character }
{ { UNLOADIN | NO UNLOADIN } }
{ VARIN }
```

out-option is:

```
{ BLOCKOUT out-block-length }
{ EBCDICOUT }
{ FOLD }
{ PAD pad-character }
{ RECOUNT out-record-length }
{ { REWINDOUT | NO REWINDOUT } }
{ SKIPOUT num-eofs }
{ { UNLOADOUT | NO UNLOADOUT } }
{ VAROUT }
{ SHAREOUT }
```

display-option is:

```
{ O[CTAL] }
{ D[ECIMAL] }
{ H[EX] }
{ BYTE }
{ A[SCII] }
{ NO HEAD }
```

move-option is:

```
{ SOURCEDICT dictionary-name }
{ SOURCEREC ddl-record-name }
{ TARGETDICT dictionary-name }
{ TARGETREC ddl-record-name }

MOVE { source-name TO target-name }
      { ( source-name TO target-name }
      { [ , source-name TO target-name]... ) }

MOVEBYNAME [ ON | OFF ]
MOVEBYORDER [ ON | OFF ]
TRUNC[ATION] [ ON | OFF ]
REDEFINE ( redefine-spec [ , redefine-spec]... ) }
```

redefine-spec is:

original-qualified-name AS *redefined-qualified-name*

in-file

is the name (or an equivalent DEFINE) of the table or file from which to copy data. *in-file* can be a table, a disk file, a labeled or unlabeled tape, a terminal, or a process.

out-file

is the name (or an equivalent DEFINE) of the table, file, tape, process, printer, spooler, or terminal to which to copy the data. If you copy to a file or table, the file or table must exist before you execute the COPY.

If you omit *out-file*, SQL uses the current OUT file.

ALLOWERRORS [ON | OFF | *num*]

specifies action when conversion errors occur.

ON Skip nonconvertible records but process subsequent records

OFF Stop the copy operation after the first conversion error

num Skip nonconvertible records until the number of such records exceeds the value of *num*. The maximum value for *num* is 32,767.

If you omit the ALLOWERRORS clause completely, the default is ALLOWERRORS OFF. If you specify ALLOWERRORS but do not specify an option, the default is ALLOWERRORS ON.

Nonconvertible records include records that contain a nonnumeric value in a numeric field, records that contain a duplicate key value in the primary key field of the output file, records that are inconsistent with constraints defined for the output table, and records that contain parity errors. For more information about rules for data conversion, see [CONVERT Command](#) on page C-94.

COUNT *num-records*

specifies the number of records to copy. The default is all records.

```
FIRST { ordinal-record-num
      { KEY record-spec
      { KEY (key-value [, key-value]... )
      { key-specifier ALTKEY (key-value
      {      [, key-value ] ... ) }
```

specifies the starting record of the input file from which to begin copying. If you omit the FIRST clause, COPY starts with the first record.

The FIRST clause is the same as the FIRST clause in the LOAD command. For information, see [LOAD Command](#) on page L-18.

REPLACE SPACES WITH { ZERO[ES] | DEFAULT[S] }

specifies how to copy an Enscribe ASCII numeric decimal field containing all blanks to an SQL numeric column. (Does not apply to Enscribe numeric binary fields)

ZEROES sets the target column to 0

DEFAULTS sets the target column to its default value

If you do not specify this option for an Enscribe ASCII numeric decimal field, a conversion error occurs for any record in which the field contains blanks.

UNSTRUCTURED

(for copying from a table or disk file only) directs COPY to handle the data as a sequence of bytes, ignoring any record structures normally recognized for the table or file. The UNSTRUCTURED option lets you examine only the one partition named as the source file in the command.

UPSHIFT

(for copying to an Enscribe file only) converts all bytes of the input that contain lowercase ASCII characters to uppercase ASCII characters before copying the data to the target record.

The UPSHIFT conversion is made without regard to the data types of fields or columns of the input, so undesired changes to the data can occur if you use UPSHIFT with input that is not composed of simple character data.

Although you cannot specify the UPSHIFT option if *out-file* is an SQL table, data moved to an SQL column that has the UPSHIFT attribute is automatically upshifted.

USESQLNULLS

(for copying from Enscribe files to SQL tables only) specifies that a null value from an Enscribe file be copied as an SQL null value.

USESQLNULLS applies only if the SQL column being copied to allows null values, if you also specify the SOURCEREC option, and if the Enscribe null value appears in every byte of the Enscribe field. (Enscribe allows you to specify a character to use as the “null character” for a field at file-creation time, then uses that character to represent null values within the field. Any field that is filled entirely with the null character is handled as null.)

If you omit USESQLNULLS, COPY provides no special treatment for Enscribe null characters.

SQLNULLABLE

specifies that the support for null indicator values for Enscribe fields is enabled when loading data from Enscribe to SQL or from SQL to Enscribe.

The SQLNULLABLE option causes COPY to look for the SQLNULLABLE DDL clause in the TARGETREC or SOURCEREC option, and depending on this clause, add or read a two-byte null indicator for the Enscribe field value. Without TARGETREC and SOURCEREC options, COPY adds or reads the null indicator for all Enscribe fields that correspond to a nullable column in the SQL table.

If you omit the SQLNULLABLE clause, COPY does not support null indicators for Enscribe fields. SQLNULLABLE does not affect loading from SQL tables to SQL

tables and Enscribe files to Enscribe files. The SQLNULLABLE option for Enscribe to SQL loading automatically enables the USESQLNULLS option.

in-option

specifies characteristics of the input file. It is identical to the *in-option* for the LOAD command. For information on *in-option* clauses, see [LOAD Command](#) on page L-18.

out-option

specifies characteristics of the output file.

BLOCKOUT *out-block-length*

(for copying to non-SQL files and processes only) specifies the number of bytes in a block of the output file (the maximum number of bytes written in a single physical operation).

out-block-length is an integer in the range 1 to 32767 that specifies a blocksize supported for *out-file*. (Not all file types support the full range of blocksizes.)

If the length of the output block is greater than the RECOU *out-record-length*, output record blocking occurs. The block is filled with *out-record-length* records until it is full or until the last output record is encountered.

If the block length is not an even multiple of *out-record-length*, the last record in a full block is truncated.

The actual number of bytes written in a physical operation is *out-block-length* for all blocks but the last one. If the last block is not full, the actual number of bytes written is equal to the number of records in the last block times the *out-record-length*.

If you omit BLOCKOUT and *out-file* is not a labeled tape, COPY uses the RECOU value for *out-block-length* and writes each output record in a separate physical operation.

If *out-file* is a labeled tape, you can specify the output block length with either the BLOCKOUT clause of the COPY command (as described here) or with the BLOCKLEN attribute of the CLASS TAPE DEFINE for the tape. If you specify values for both the BLOCKOUT clause and the BLOCKLEN attribute, the values must match.

EBCDICOUT

(for copying to non-SQL files and processes only) translates ASCII characters to their EBCDIC equivalents in the output file. If you omit EBCDICOUT, COPY does not translate output.

In a conversion between ASCII and EBCDIC, the symbols representing each character are the same in ASCII and EBCDIC except for:

ASCII	EBCDIC
Exclamation point	Logical OR
Left square bracket	Cent sign
Right square bracket	Exclamation point
Circumflex	Logical NOT sign

The conversion is done without regard to the data types of fields or columns of the input, so undesired changes to the data can occur if you use EBCDICOUT with input that is not composed of simple character data.

FOLD

(for copying to Enscribe files only) divides input records that are longer than *RECOUNT out-record-length* into as many *out-record-length* records as needed to copy the entire input record. If the last record written is shorter than *out-record-length* because of a FOLD, and you specify PAD, padding occurs. If you omit FOLD, truncation occurs when an input record is longer than the output file's record length.

PAD *pad-character*

(for copying to Enscribe files only) pads output records that contain less than *out-record-length* bytes with *pad-character*, up to the record length specified in the file label of the output file. Specify *pad-character* as a single ASCII character inside quotation marks ("c") or as a numeric literal in the range 0 through 255, representing the byte value of the character.

RECOUNT *out-record-length*

(for copying to Enscribe files or tapes only) specifies the maximum length of an output record in bytes. The actual number of bytes written for each output record (the write count) depends on whether you also specify PAD:

- If you do not specify PAD, the write count is either the read count or *out-record-length*, whichever is less.
- If you specify PAD, the write count is *out-record-length*.

In either case, if the number of input bytes exceeds *out-record-length*, the input record is truncated at output-record length bytes (unless you specify FOLD).

- If you omit RECOUNT and *out-file* is not a labeled tape, COPY determines the *out-record-length*:
 - If you specify *out-block-length* as less than or equal to 4096, the value of *out-block-length* is used for *out-record-length*. If

out-block-length is greater than 4096, *out-record-length* is 4096.

- If you do not specify *out-block-length* and if *out-file* is an unstructured disk file, if *in-file* is an SQL table, VAROUT is not specified, and no display option was specified, *out-record-length* is the length of the logical record specified by TARGETREC, or—if TARGETREC is not specified—the length of the logical record implied by the description of the input table. Otherwise, *out-record-length* is 132.
- If you do not specify *out-block-length* and if *out-file* is a process file, *out-record-length* is 132.
- If you do not specify *out-block-length* and if *out-file* is a structured disk file or a nondisk device, *out-record-length* is the record length specified or assigned when the file is created (or when the system is generated).

If *out-file* is a labeled tape, you can specify the output record length with either the RECOUNT clause of the COPY command (as described here) or with the RECLEN attribute of the CLASS TAPE DEFINE for the tape. If you specify values for both the RECOUNT clause and the RECLEN attribute, the values must match.

{ REWINDOUT | NO REWINDOUT }

(for copying to magnetic tapes only) specifies whether to rewind the tape when COPY completes. The default is REWINDOUT (the tape is rewound).

SKIPOUT *num-eofs*

(for copying to unlabeled magnetic tapes only) moves the tape past *num-eofs* end-of-file (EOF) marks before starting to copy the data. Specify *num-eofs* as an integer from -255 through 255.

If you specify a positive value for *num-eofs*, the tape is wound forward past *num-eofs* EOF marks and is positioned immediately after the last EOF mark passed.

If you specify a negative value for *num-eofs*, the tape is wound backwards over (-1 times *num-eofs*) EOF marks, moved forward so that it is positioned immediately ahead of the last EOF mark passed.

If you specify a value of 0 for *num-eofs*, the SKIPOUT option is ignored.

If you omit the SKIPOUT option, the tape remains at its current position, and data transfer begins with the next physical record on tape.

```
{ UNLOADOUT | NO UNLOADOUT }
```

(for copying to magnetic tapes only) specifies whether the tape is unloaded when rewinding occurs. The default is UNLOADOUT (the tape is unloaded when it is rewound).

VAROUT

(for copying to Enscribe files only) writes variable-length, blocked records.

Each Enscribe variable-length record is preceded by a one-word indicator containing the record length in bytes. The indicator is always aligned on a word boundary, although records might contain an odd number of bytes. The indicator and the write count are equal although the record might have been truncated. Truncation occurs if the record is longer than *out-record-length* or *out-block-length* minus two. (Two extra bytes are required for the indicator.)

Records cannot span blocks. If the next record with its indicator does not fit into the current block, VAROUT terminates the current block and begins a new block.

VAROUT terminates a block by writing a 1-word block terminator of -1 (%177777) to indicate that no more valid records are in the block, then padding the remainder of the physical block. VAROUT cannot write the terminator when the previous record ends on a block boundary or when *out-block-length* is odd and only one byte remains in the block.

Empty or zero-length records are supported.

The PAD and FOLD options are not allowed with VAROUT.

This sample block has a BLOCKOUT length of %30. The three records "FRESNO", "MUNICIPAL", and "BANK" illustrate the action of VAROUT:

%000006	<-	Length indicator for record 1
F R	<-	
E S		Record 1
N O	<-	
%000011	<-	Length indicator for record 2
M U	<-	
N I		Record 2
C I		
P A	<-	
L	<-	
%177777	<-	Block terminator (end of block)
p p	<-	Padding

For the third record, BANK and its record-length indicator require six bytes, beginning on a word boundary. Because only four bytes remain in the sample block, VAROUT terminates the block and writes the BANK record to the next block.

SHAREOUT

opens the output file with SHARED access. If you omit SHAREOUT, the output file is opened with EXCLUSIVE access.

display-option

(for copying to Enscribe files only) specifies the format for displaying the file:

O[CTAL]	Display in octal and ASCII
D[ECIMAL]	Display in decimal and ASCII
H[EX]	Display in hexadecimal and ASCII.
BYTE	Display in byte format and ASCII, convert each byte separately
A[SCII]	Display in ASCII
NO HEAD	Omit the heading preceding each record

If you do not specify BYTE, COPY handles each word as a single value and converts it accordingly. If you specify BYTE but not OCTAL, DECIMAL, or HEX, the display appears in byte-octal format.

If you specify more than one of OCTAL, DECIMAL, and HEX, each line is displayed in each specified format in the order: octal, decimal, and hexadecimal.

The ASCII option has no meaning when combined with OCTAL, DECIMAL, HEX, or BYTE.

move-option

specifies names of elements related to the table or Enscribe file and how to map source names to different target names. *move-option* is identical to *move-option* for the LOAD command. For information on *move-option* clauses, see [LOAD Command](#) on page L-18 and for considerations related to move options, see [Considerations—LOAD](#) on page L-33.

Considerations—COPY

- COPY requires authority to read the source file and authority to write to the target file. If you are copying to or from a table, you must also have authority to read the catalog in which the table is described.
- COPY performs these operations:
 - From an Enscribe file to a table—each source record from the file is inserted as a row in the target table. Each elementary field value in the source record is converted into a column value in the table row generated from the source record. Values are also copied to the indexes of the table.
 - From a table to an Enscribe file—each row from the table is added as a record to the target file. Each column value in the source row is converted to an elementary field value in the target record.

- From one table to another—each source row is inserted as a row in the target table.
- You might want to use COPY instead of LOAD for these reasons:
 - You can copy data within a user-defined TMF transaction.
 - You can copy data to an unstructured file or a nondisk file.
 - You can append or insert data without erasing existing data.
- These rules govern the transfer of data across character sets. A COPY that violates these rules terminates with an error.

Source and Target File Types	Source Field Character Set	Target Field Character Set
SQL to SQL	UNKNOWN	Any character set
	ISO88591	ISO88591
	ISO88599	ISO88599
	KANJI	KANJI
	KSC5601	KSC5601
SQL to Enscribe	UNKNOWN	PIC X or PIC N
	ISO88591	PIC X
	ISO88599	PIC X
	KANJI	PIC N
	KSC5601	PIC N
Enscribe to SQL	PIC X	Any character set
	PIC N	Any character set

For example, if the source field character set is UNKNOWN, you can copy it to a target field associated with any character set. If the source field character set is one of the nine supported ISO character sets, you can copy it only to a target field associated with that same character set.

In addition, if you copy double-byte data into a single-byte field or copy single-byte data into a double-byte field, the target field must be the same length, in bytes, as the source field.

(Enscribe-to-Enscribe copies do no field-by-field conversion, so that case is not shown in the previous table.)

- You can use COPY to display the contents of a table or Enscribe file on a terminal or printer. For example, this command prints ten rows of the EMPLOYEE table:

```
>> COPY PERSNL.EMPLOYEE, $SYS1.#PRINTER, ASCII COUNT 10;
```

The display includes the file name, the ordinal number of each row or record, the length of each record (in decimal bytes), and the ASCII representation of each line.

- If you use COPY to write to an audited file or table, the write always takes place within a TMF transaction. COPY starts a transaction if a user-defined transaction is not in progress. Because copying large amounts of data results in large amounts of TMF audit information, you might want to use ALTER TABLE to turn off the AUDIT attribute of the target file before the COPY and reset it after the COPY.

If you use COPY to write to a nonaudited file or table within a user-defined transaction, COPY issues a warning but performs the COPY anyway.

You can press the Break key to interrupt COPY. If the target file or table is audited, the COPY is rolled back and all the work is undone. If the target is nonaudited, all the work done by COPY up to the point of the break is committed.

If COPY fails and the target table is audited, TMF performs the recovery operation. If the target table is nonaudited, the data might be partially copied.

- COPY does not overwrite existing data. If *out-file* has key-sequenced file organization, COPY inserts data at the appropriate locations in the table or file, as determined by the key. COPY reports an error if a record already exists that has the same key as a record to be inserted. If *out-file* has entry-sequenced, relative, or unstructured file organization, COPY appends data to the end of the file.
- When you enter a COPY command without the SHARE option, COPY opens the input file in read-only access mode and protected exclusion mode (unless the source file is a terminal, which is always opened with shared exclusion mode). If, however, you include the SHARE option in the command when copying a disk file, COPY opens the file with shared exclusion mode.

By default, COPY opens the output file with protected exclusion mode unless the file is a terminal.

- If you copy a table or file that contains data records and zero-length (empty) records to a table or file with relative file organization, all records are written with this exception: if the *in-file* also has relative file organization, the empty records are skipped unless you specify NO COMPACT.

For example, when you copy a table or file with relative file organization that contains a combination of eight data records and two empty records, *out-file* file has eight records instead of ten. Thus, if you copy empty records from a table or file with relative file organization to another table or file with relative file organization, you lose the empty records. To transfer empty records from a table or file with relative file organization, include the NO COMPACT option.

- If the target is a table, using the COPY command is equivalent to using a set of INSERT commands with STABLE ACCESS and APPEND options. Values are inserted in the indexes of the table automatically. All the conditions that must be satisfied for an INSERT must be satisfied for each target row.

When a source column is undefined and the target column is defined with the NO DEFAULT clause, an error occurs. The source row must supply a value for every column of the target row that is defined with the NO DEFAULT clause. In addition, a target row must satisfy any constraint defined on the table to be inserted in the table. The TRUNC option determines whether values are truncated. For more information, see [INSERT Statement](#) on page I-14.

Any column except a system-defined primary key can be a source or target item.

- Rules for using CLASS TAPE DEFINES or labelled tapes are described in the discussion of the FUP COPY command in the *File Utility Program (FUP) Reference Manual*.

You cannot copy a multireel set of tapes created by a COBOL program, because COBOL does not mark the end of a multireel set in the same way that COPY does. You must use a COBOL program to copy such tapes. For more information, see the *COBOL Reference Manual*.

- Using COPY with non-SQL files
 - If the source is an EDIT file, COPY handles it like a structured file: each text line is handled as a logical record with a read count attribute. (This differs from the usual treatment of unstructured files in which each physical read, except possibly the last read of a file, returns exactly *in-record-length* bytes.) If an EDIT file record ends in the middle of a field, SQL adds blanks to the target file for the remainder of the field.
 - Although you can specify a block size up to 32,767 bytes for the BLOCKIN and BLOCKOUT parameters, some peripheral devices have smaller maximum block sizes that must not be exceeded when using the COPY command.
 - Be careful when you use PAD and TRIM options. If your data contains *trim-character* or *pad-character*, data might be added or lost. Use a pad character or trim character that is not contained in your data. For example, suppose that you pad each record of a file with zeros to a standard size in bytes and then store the records in another file. If you later trim the trailing zeros when you COPY the stored records, zeros at the end of the original data are trimmed.
- Rules for enscribe files with variable-length records

Copying Into SQL Tables

- The target file must have default values defined for columns that do not have source fields mapped to them and for all fields missing from the source record. A column must be defined with the DEFAULT clause of the CREATE TABLE or ALTER TABLE command. A DDL data item must be defined with the VALUE IS clause.
- If a record from a non-SQL source is not long enough to supply data to all fields mapped to target columns, each target column whose source field is missing must have a default value defined for it.

- In general, if a source record from a non-SQL source does not end exactly at a field boundary, an error occurs. These exceptions apply:
 - If the record ends in the middle of a VARCHAR field, the end of the record defines the end of the VARCHAR data.
 - If the file is an EDIT file and the record ends in the middle of a field, SQL adds enough blanks to the end of the input record to fill the field. In such a case, blanks must be acceptable in that column of the source record. For example, a DECIMAL column would not accept blanks; a CHAR column would.
- If the record contains an array defined by an OCCURS DEPENDING ON clause and at least one element of the array is present, the field that contains the count must be present and the number of elements in the record must be equal to the value of the field that contains the count.

Enscribe Field Formats

If an Enscribe file is the source or target, COPY copies only fields whose DDL definitions conform to these rules:

- The field must be elementary unless it is the special DDL group that represents a variable-length character string, in which case it is handled as one field during the COPY operation. This DDL group has this structure and is converted to a column with data type VARCHAR:

```
02 A-VARCHAR.
   03 LEN PIC S9(4) COMP.
   03 VAL PIC X(len).
```

- The field must not be a FILLER field.
- COPY ignores these DDL clauses:
 - Level-88 CONDITION-NAME clause
 - Level-66 RENAMES clause
- Unless you specify a REDEFINES clause in the REDEFINE option of the COPY command, COPY ignores the clause and uses the original field definition.

Field Conversions

For any COPY operation, the data type of each source field must be compatible with the data type of its corresponding target field. The details of data type compatibility and Enscribe-to-SQL and SQL-to-Enscribe field conversions are identical for COPY and LOAD; For information, see [Data type compatibility and field conversions](#) on page L-36.

Examples—COPY

- Suppose that in addition to the table EMPLOYEE described in the catalog \$VOL1.PERSNL, you have created an identical table EMPLOYEE described in a catalog named \$VOL2.TESTC. You must qualify the table names enough to identify the location of each one uniquely. COPY determines which catalogs to use, such as,

```
>> COPY $VOL1.PERSNL.EMPLOYEE, $VOL2.TEST.EMPLOYEE;
```

If the default subvolume is \$VOL1.PERSNL, this command performs the same function:

```
>> COPY EMPLOYEE, $VOL2.TEST.EMPLOYEE;
```

The tables have identical descriptions. You need not specify any move options; MOVEBYORDER is used by default.

- This command displays the first 100 rows of table \$VOL1.SALES.PARTS at your terminal and specifies the octal display format:

```
>> COPY $VOL1.SALES.PARTS, ,OCTAL COUNT 100;
```

- This example demonstrates the COUNT and MOVEBYNAME options. The table SPAREPRT contains columns of the same names as the PARTS table, but the columns in the two tables are arranged in a different order. This copies the first 300 rows from \$VOL1.SALES.PARTS table to \$VOL1.SALES.SPAREPRT table and copies each column in the source table to the column with the same name in the target table:

```
>> COPY $VOL1.SALES.PARTS, $VOL1.SALES.SPAREPRT,
+> COUNT 300 MOVEBYNAME;
```

- This command copies data from the Enscribe file \$VOL2.SUBV1.CLIENTS to the table \$VOL1.SALES.CUSTOMER:

```
>> COPY $VOL2.SUBV1.CLIENTS, $VOL1.SALES.CUSTOMER,
+> SOURCEDICT $VOL2.SUBV1, SOURCEREC CLREC,
+> MOVEBYORDER, TRUNC ON;
```

Because the source is an Enscribe file, the command includes a SOURCEREC option to identify the DDL record definition for the Enscribe file. The values are copied in order from fields to columns, as shown:

Fields in Record CLREC	Columns in Table CUSTOMER
02 CUSTNUM PIC 9(4).	CUSTNUM DECIMAL(4) UNSIGNED,
02 CUSTNAME PIC X(20).	CUSTNAME CHAR(18),
02 ADDR.	
03 ADDRESS PIC X(22).	STREET CHAR(22),
03 CITY PIC X(14).	CITY CHAR(14),
03 STATE PIC X(2).	STATE CHAR(12),
03 ZIP-CODE PIC X(6).	POSTCODE CHAR(10),
	CREDIT CHAR(2) DEFAULT "C1"

The CUSTNAME value is truncated because the column length is less than the field length. The default value is used for the CREDIT column in each new row of CUSTOMER because no source field maps to this column.

- This example demonstrates the MOVE option. This command copies data from the table \$VOL1.SALES.PARTS to the Enscribe file \$TESTVOL.SALES.PARTS, copying only the PARTNUM and PARTDESC columns to the PARTNUMR and PARID columns respectively:

```
>> COPY $VOL1.SALES.PARTS, $TESTVOL.SALES.PARTS,
+> TARGETDICT $TESTVOL.SALES, TARGETREC DFORMAT,
+> MOVE (PARTNUM TO PARTNUMR, PARTDESC TO PARTD);
```

A target column that does not have a source column mapped to it receives its default value, unless no default value is defined for it, in that case COPY returns an error.

Because the target is an Enscribe file, the command includes a TARGETREC option to identify the DDL record definition for the Enscribe file.

- This example demonstrates the FIRST KEY option, provided that a table with a two-column key as described by this CREATE TABLE statement exists:

```
>> CREATE TABLE EMP (
+>   EMPNUM SMALLINT, EMPNAME VARCHAR (20),
+>   SALARY SMALLINT, PRIMARY KEY (EMPNUM, EMPNAME) );
```

The COPY command displays rows of table EMP at your terminal in hexadecimal format, starting with the row for the employee whose employee number is 100 and whose name is Martin Smith:

```
>> COPY EMP, ,HEX, FIRST KEY (0, 100, "MARTIN SMITH  ");
```

The EMPNUM column is defined as a SMALLINT, which uses two bytes of storage. For an employee number of 100, the first byte of the EMPNUM column contains 0 and the second byte contains 100. The second byte contains 100 because the data is shifted to the right. For the EMPNAME column, the key-value must be fully padded to 20 characters, which is the maximum length of the VARCHAR used to define the column.

Correlation Names

A correlation name is a name associated with a table or view in an SQL statement for one or more of these reasons:

- To distinguish the table or view from another table or view referred to in the statement
- To qualify an ambiguous column reference
- To distinguish different uses of the same table
- To make the query shorter

A correlation name can be explicit or implicit.

An explicit correlation name is an SQL identifier associated with a table or view in the FROM clause of a SELECT statement, in the *select-statement* of an INSERT statement, or in a subquery. The name must be unique within the FROM clause.

An explicit correlation name is known only to the statement in which you define it. You can use the same identifier as a correlation name in another statement.

A table or view reference that has no explicit correlation name has an implicit correlation name. The implicit correlation name is the table or view name without the optional subvolume, volume, and node qualifiers, or—if the reference to the table or view is a DEFINE—the portion of the DEFINE name that follows the equals sign (=).

You cannot use an implicit correlation name for a reference that has an explicit correlation name within the statement.

This example shows the uses of both explicit and implicit correlation names. The query refers to two tables (ORDERS and CUSTOMER) that contain columns named CUSTNUM. In the WHERE clause, one column reference is qualified by an implicit correlation name (ORDERS) and the other by an explicit correlation name (X):

```
SELECT ORDERNUM, CUSTNAME FROM ORDERS, CUSTOMER X
WHERE ORDERS.CUSTNUM = X.CUSTNUM and ORDERS.CUSTNUM = 543;
```

COUNT Function

COUNT is a function that counts the number of rows that result from a query or the number of rows that contain a distinct value in a specific column.

The result of COUNT is data type LARGEINT. The result can never be null.

COUNT { (*) { (DISTINCT <i>column-name</i>) }

*

specifies that COUNT should not exclude null values from the aggregate set. If the set is empty, COUNT returns zero.

DISTINCT *column-name*

specifies a set of distinct column values from each row of the result table to determine COUNT. The column cannot be a column from a view that corresponds to an expression in the view definition.

Duplicate rows are eliminated only if you specify DISTINCT; otherwise, all rows are included whether or not you specify ALL.

If you specify DISTINCT in more than one COUNT function in the same statement, the functions must refer the same column.

Specifying DISTINCT with the COUNT function places no restrictions on the use of DISTINCT with AVG, SUM, MAX, or MIN.

Considerations—COUNT

- COUNT is evaluated after eliminating all null values from the aggregate set, unless you specify an asterisk (*). If the set is empty, COUNT returns zero.

Example—COUNT

This example counts the number of distinct departments:

```
>> SELECT COUNT (DISTINCT DEPTNUM) FROM PERSNL.EMPLOYEE;
      (EXPR)
-----
                11
--- 1 row(s) selected.
>>
```

CPRLSRCE Table

The CPRLSRCE table is a catalog table that contains source definitions for each collation described in the CPRULES table. [Table C-9](#) lists the contents of the CPRLSRCE table.

Table C-9. The CPRLSRCE Table

Column Name	Data Type	Description
1 CPRULESNAME *	CHAR(34)	Collation name
2 SEQNUMBER *	SMALLINT UNSIGNED	Sequence number for the source line; first line has SEQNUMBER = 1
3 TEXT	VARCHAR (256)	Source text

* Indicates primary key

The CPRLSRCE table was added in version 300.

Guardian names in the CPRLSRCE table are fully qualified and use uppercase characters.

CPRULES Table

The CPRULES table is a catalog table that contains one row for each collation. [Table C-10](#) describes the contents of the CPRULES table.

Table C-10. The CPRULES Table

Column Name	Data Type	Description
1 CPRULESNAME *	CHAR(34)	Collation name
2 CHARACTERISTICS	CHAR(1)	Properties of the collation: O, if only identical strings sort equal N, if some nonidentical strings sort equal
3 CPRULESCCLASS	CHAR(1)	Always U.
4 CPROBJSIZE	INTEGER UNSIGNED	Minimum size of buffer needed to store object
5 CHARACTERSET	CHAR (30)	Name of character set assumed by the collation
6 CPRULESVERSION	SMALLINT UNSIGNED	Version number of most recent version feature used in collation

* Indicates primary key

The CPRULES table was added in version 300.

Guardian names in the CPRULES table are fully qualified and use uppercase characters.

CREATE CATALOG Statement

CREATE CATALOG is a DDL statement that creates a new catalog. Each new catalog includes a complete set of catalog tables and indexes. For information about the contents of a catalog, see [Catalogs](#) on page C-8.

```
CREATE CATALOG [ catalog ] [ SECURE "rwep" ]
               [ PHYSVOL volume-name ] [ attribute-spec ]
```

attribute-spec is:

```
{ EXTENT { (pri-ext-size[,sec-ext-size]) } }
  { ext-size }
{ MAXEXTENTS integer }
```

Note. The *attribute-spec* option is supported on systems running J06.05 and later J-series RVUs or H06.16 and later H-series RVUs.

catalog

is the name of the Guardian subvolume to contain the catalog (or an equivalent DEFINE) and is also the name of the new catalog. The volume on which the subvolume resides must be audited by the TMF subsystem and *catalog* must be a unique catalog name on that volume. If SMF is installed on your node, the volume on which the subvolume resides can be a virtual or direct volume. You cannot specify any *\$.ZYS** subvolumes for *catalog*.

If SMF is installed on your node and the default volume is a virtual volume, SQL places the set of catalog tables on the virtual volume. If the default volume is a direct volume, the catalog tables reside on the physical volume as direct files not managed by SMF.

If you omit *catalog*, SQL uses the current default catalog.

SECURE "*rwep*"

specifies security for the new catalog. If you omit the SECURE clause, SQL uses the default security of the user who creates the catalog. (For more information, see [Security](#) on page S-11.)

PHYSVOL *volume-name*

If SMF is installed on your node, specifies a physical volume on which to place the set of catalog tables. This option overrides SMF features. *volume-name* can be either the name of a physical volume or equivalent DEFINE. Do not include the node name in your volume name.

This option is available only if you specify a virtual volume and subvolume for *catalog*. *volume-name* must belong to the virtual volume you specify.

`attribute-spec`

specifies physical file attributes for the file that holds catalog tables and indexes.

The following file attributes can be specified through the `attribute-spec` option:

```
EXTENT { (pri-ext-size [, sec-ext-size]) }
        { ext-size }
```

The various options allowed under *ext-size*, *pri-ext-size*, or *sec-ext-size* are:

```
integer[ PAGE[S]          ]
        [ BYTE[S]         ]
        [ REC[S]          ]
        [ MEGABYTE[S]     ]
        [ GIGABYTE[S]     ]
```

For Catalog tables, the default is 16 pages for the primary extent and 128 pages for each secondary extent.

For Catalog indexes, the default is 16 pages for the primary extent and 64 pages for each secondary extent.

The default unit type is PAGE.

For more information see, [EXTENT File Attribute](#) on page E-30

`MAXEXTENTS num-extents`

is an integer from 1 to 940 that specifies the maximum number of extents that can be allocated.

The catalog tables and indexes are of Format 1 type.

For more information see, [MAXEXTENTS File Attribute](#) on page M-2.

Note. The `attribute-spec` option is supported on systems running J06.05 and later J-series RVUs, or H06.16 and later H-series RVUs.

Considerations—CREATE CATALOG

- CREATE CATALOG requires authority to write to the SQL.CATALOGS table, because SQL adds an entry to that table for the new catalog.

The owner of the new catalog is the user whose process created the catalog. However, the operations allowed on the tables and indexes that make up the catalog itself (as described under [Catalogs](#) on page C-8) are more limited than those allowed on ordinary tables and indexes, even for the owner. You can delete catalog tables only with DROP CATALOG (not even with CLEANUP unless you specify CLEANUP *,CATALOG;), you cannot divide catalog tables, and you cannot alter file attributes of catalog tables except for those related to security.

Secure catalogs so that other users who need to access them have appropriate authority. Users who require write access must have read access as well.

Programs that use objects described in a catalog must have write access to the TRANSIDS and USAGES tables. For programs to be registered in a catalog, the SQL-compiling process must have write access to the PROGRAMS, TRANSIDS, and USAGES tables.

Catalogs can be resecured with ALTER CATALOG. The PROGRAMS, TRANSIDS, and USAGES tables (but not other catalog tables) can be individually resecured with ALTER TABLE.

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 execute a CREATE CATALOG statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. (For information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.)

- For better performance when several SQL catalogs are on the same disk volume, the system administrator should set the disk process cache to an appropriate value with the Peripheral Utility Program (PUP) SETCACHE command. This strategy is especially important for 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 the operation to take up to 25 times longer than if you set disk cache to 4 MB.

For more information about managing cache, see the *SQL/MP Installation and Management Guide*. For information about PUP for D-series systems, see the *Peripheral Utility Program (PUP) Reference Manual*.

If SMF is installed on your node, there are two ways to place the set of catalog tables on a single physical volume:

- Specify a direct subvolume for *catalog*
- Specify a virtual volume and subvolume for *catalog* and a physical volume that belongs to the virtual volume in PHYSVOL. Do not include node name in your volume name.

If you specify a virtual volume for *catalog* and omit the PHYSVOL option, SQL can distribute catalog tables among multiple physical volumes in the virtual volume.

- The version number of a new catalog is the version of NonStop SQL/MP on the node where the catalog resides, even if you create the catalog from a node with a different version number.

For example, if you issue CREATE CATALOG from a version 315 node but specify a *catalog* subvolume on a version 1 node, the new catalog is a version 1 catalog.

- The user specified EXTENT and MAXEXTENTS attributes apply for catalog tables and indexes.
- If the user specifies the EXTENT attribute with only primary extent value, then the secondary extent value is also set with the same value as primary extent.
- If the user specifies the EXTENT attribute but does not specify the MAXEXTENTS attribute, then MAXEXTENTS will be calculated by SQL, based on the primary and secondary extents value.
- If the user specifies the EXTENT attribute and MAXEXTENTS attribute, then SQL will check if the user specified MAXEXTENTS is lower than the maximum possible extents based on the primary and secondary extents value. If it is lower, SQL accepts the user specified MAXEXTENTS. If it is higher, an error is returned during file creation.

Examples—CREATE CATALOG

- This example creates a catalog named PERSNL on node \SYS1 and volume \$VOL1, with security "nunu":

```
CREATE CATALOG \SYS1.$VOL1.PERSNL SECURE "nunu";
```

- This SQLCI example uses ALTER DEFINE to set the CATALOG attribute of the =_DEFAULTS DEFINE before creating a catalog. The new catalog is created on \SYS1.\$VOL.SALES.

```
ALTER DEFINE =_DEFAULTS, CATALOG \SYS1.$VOL.SALES;
CREATE CATALOG;
```

- This example creates a catalog named PERSNL on node \SYS1 and volume \$VOL1, with EXTENT 100. MAXEXTENTS will be calculated by SQL:

```
CREATE CATALOG \SYS1.$VOL1.PERSNL EXTENT 100;
```

- This example creates a catalog named PERSNL on node \SYS1 and volume \$VOL1, with EXTENT (100,200). MAXEXTENTS will be calculated by SQL:

```
CREATE CATALOG \SYS1.$VOL1.PERSNL EXTENT (100,200);
```

- This example creates a catalog named PERSNL on node \SYS1 and volume \$VOL1, with EXTENT (100,200) and MAXEXTENTS 150:

```
CREATE CATALOG \SYS1.$VOL1.PERSNL EXTENT (100,200) MAXEXTENTS  
150;
```

CREATE COLLATION Statement

CREATE COLLATION is a DDL statement that creates a collation.

```
CREATE COLLATION name { FROM source
                        { LIKE coll [ WITH COMMENTS ] }
  [ CATALOG catalog      ] [ PHYSVOL volume-name ]
```

name

is a Guardian name (or an equivalent DEFINE) that is the name of the new collation.

If SMF is installed on your node, the volume portion of *name* can be a virtual or direct volume. If you specify only a subvolume, SQL creates a new collation object in the current default volume. If the default volume is virtual, the collation resides on the virtual volume. If the default volume is direct, the collation resides on the physical volume as a direct file not managed by SMF.

FROM *source*

directs SQL to create the new collation by calling the collation compiler to compile the definition in the EDIT file *source* (or an equivalent DEFINE). For information about the contents of *source*, see [Collation Definitions](#) on page C-30.

If the compilation fails, SQL returns an error and does not update the catalog or create the collation. The error message includes the name of a file that contains diagnostic information about the compilation. For information about the diagnostic, see the *SQL/MP Messages Manual*.

LIKE *coll* [WITH COMMENTS]

directs SQL to create the new collation like an existing collation *coll*. *coll* is a collation name or an equivalent DEFINE. SQL does not include comments from *coll* in the new collation unless you specify WITH COMMENTS.

CATALOG *catalog*

is the name of the catalog (or an equivalent DEFINE) in which to register the new collation. If you omit the CATALOG clause, SQL uses the current default catalog.

PHYSVOL *volume-name*

If SMF is installed on your node, the PHYSVOL option directs SQL to override SMF and place the collation object on the physical volume *volume-name*. For *volume-name*, specify either a physical volume or equivalent DEFINE. Do not include the node name in your volume name.

This option is available only if you specify a virtual volume for *name*. *volume-name* must belong to the virtual volume you specify.

Consideration—CREATE COLLATION

CREATE COLLATION requires read and write authority for the catalog in which the new collation is registered. The FROM clause requires read authority for the *source* file. The LIKE clause requires read authority for the existing collation *coll* and for the associated catalog tables.

Example—CREATE COLLATION

This example statement creates a collation named TRANSL2 from the definition in the EDIT file \$DATA.COLL.TRANSL2:

```
CREATE COLLATION TRANSL2 FROM $DATA.COLL.TRANSL2;
```

CREATE CONSTRAINT Statement

CREATE CONSTRAINT is a DDL statement that defines a constraint for a table. When a constraint is in effect, all rows in the table, either directly or through a view, must satisfy the constraint.

```
CREATE CONSTRAINT constraint ON table

    CHECK condition [DEFERRED]
```

constraint

is the name of the constraint. *constraint* must be an SQL identifier that is unique for the associated table. If SMF is installed on your node, the name of the associated table must be either a virtual or direct name.

ON *table*

specifies the table associated with the constraint (or an equivalent DEFINE).

If *table* is a partition, the constraint applies to the entire table to which the partition belongs. To create a constraint that applies only to a specific partition, include the range of key values as part of the CHECK clause criteria. For example, a constraint on a partition of the PARTLOC table in the sample database could include this clause:

```
CHECK LOC_CODE  >= "G00" AND LOC_CODE < "P00"
```

CHECK *condition*

is a search condition that specifies the conditions of the constraint and that is satisfied by all existing rows of *table*. The search condition must follow these rules:

- The text of the condition must have fewer than 3,000 bytes.
- The combined search conditions of all constraints associated with a table must have fewer than 31,000 bytes.
- The search condition cannot include a function other than UPSHIFT, and cannot include a subquery, a host variable, or a system-created SYSKEY column.
- For any row of *table*, the search condition must be resolved by looking only at that row.

DEFERRED

is an option to delay the validation of the constraint against all the existing rows in the table. The constraint is applied immediately to the table to validate the new inserts or updates. The locks on the table are released after the constraint is applied. Then a browse-mode validation of the existing rows is completed. This

enables concurrent DML operations on the table to complete without waiting behind the locks.

Considerations—CREATE CONSTRAINT

- To create a constraint, you must be a generalized owner of the underlying table. You must also have authority to read the table and authority to write to affected catalogs.

CREATE CONSTRAINT requires an exclusive open on *table*, including any partitions. The operation fails if the table is inaccessible or if other users have the table open.

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 execute a CREATE CONSTRAINT statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. (For information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.)

- When you create a constraint, SQL adds it to those constraints that already exist for the table. The new constraint affects later INSERT and UPDATE operations; it does not affect existing constraints.

You can determine the existing constraints for a table by querying the CONSTRNT table of the catalog that contains the table description.

To cancel a constraint, use the DROP statement.

- CREATE CONSTRAINT fails if the table contains data that violates the constraint being created.
- The DEFERRED option is not supported in a user-defined TMF transaction.

When your table is large, you might want to avoid executing CREATE CONSTRAINT in a user-defined TMF transaction. For a large table, the CREATE CONSTRAINT operation might run for a long time. The delay could cause TMF to require too much log file space to perform the logging required for all users.

If no user-defined TMF transaction is in progress when CREATE CONSTRAINT executes, SQL automatically starts several separate transactions during the operation. None of these transactions spans the entire lengthy period during which the table is tested for qualifying rows.

- When a CREATE CONSTRAINT with the DEFERRED option is used, the constraint is applied on the table and a browse-mode validation of the existing rows is performed. In this window, certain inserts or updates on the table fail if the constraint is violated. Later, if the browse-mode validation of the existing rows does not satisfy the constraint, it will be dropped. As a result, the inserts or updates fail due to the nonexistent constraint.
- CREATE CONSTRAINT invalidates SQL programs that use the underlying table.

Examples—CREATE CONSTRAINT

- This example creates a constraint to ensure that only values greater than \$10,000 are entered in the SALARY column:

```
CREATE CONSTRAINT ASAL ON \SYS1.$VOL1.PERSNL.EMPLOYEE
    CHECK SALARY > 10000;
```

- This example creates a constraint that enforces a relationship between two items in a row. In this case, the constraint ensures that a delivery date for an order is not earlier than the date the order was taken:

```
CREATE CONSTRAINT DATE_CONSTRNT ON SALES.ORDERS
    CHECK DELIV_DATE >= ORDER_DATE;
```

- This example creates a constraint to ensure that data in a character column is stored in uppercase letters:

```
CREATE CONSTRAINT UPSHIFT_DESCRIPTION ON PARTS
    CHECK PARTDESC = UPSHIFT (PARTDESC);
```

CREATE INDEX Statement

CREATE INDEX is a DDL statement that creates an index based on one or more columns of a table.

```
CREATE [ UNIQUE ] INDEX index ON table

    ( col [ ASC[ENDING] ] [ collate-spec ]
      [ DESC[ENDING] ] ]

    [ , col [ ASC[ENDING] ] [ collate-spec ] ] ... )
      [ DESC[ENDING] ] ]

[ | CATALOG catalog | ]
[ | PHYSVOL volume-name | ]
[ | { INVALIDATE | NO INVALIDATE } | ]
[ | KEYTAG key-specifier | ]
[ | PARALLEL EXECUTION { ON [ CONFIG file ] | OFF } | ]
[ | PARTITION ( partition [ , partition ] ... ) | ]
[ | WITH SHARED ACCESS [ wsa-spec ] | ]
[ | attribute-spec | ]
```

collate-spec is:

```
COLLATE { collation | CHARACTER SET }
```

wsa-spec is:

```
{ | NAME operation-name | }
{ | REPORT [ TO collector | ON | OFF ] | }
{ | { COMMIT [ WORK ] commit-options } | }
{ | { ROLLBACK [ WORK ] } | }
```

attribute-spec is:

```
{ | ALLOCATE integer | }
{ | { AUDITCOMPRESS | NO AUDITCOMPRESS } | }
{ | BLOCKSIZE integer | }
{ | { BUFFERED | NO BUFFERED } | }
{ | { CLEARONPURGE | NO CLEARONPURGE } | }
{ | { DCOMPRESS | NO DCOMPRESS } | }
{ | DSLACK percent | }
{ | EXTENT { size | (pri-size [, sec-size]) } | }
{ | { FORMAT 1 | FORMAT 2 } | }
{ | { ICOMPRESS | NO ICOMPRESS } | }
{ | ISLACK percent | }
{ | LOCKLENGTH integer | }
{ | MAXEXTENTS integer || { | NOPURGEUNTIL date | } | }
{ | { SERIALWRITES | NO SERIALWRITES } | }
{ | SLACK percent || { | TABLECODE integer | } | }
{ | { VERIFIEDWRITES | NO VERIFIEDWRITES } | }
```

UNIQUE

specifies that values in the column or set of columns that make up the index field cannot be the same for two or more rows of the table. For indexes with multiple columns, the value of the columns as a group determines uniqueness, not the values of the individual columns.

SQL cannot create a UNIQUE index if any *col* specified for the index is a column that allows null values or if the underlying table has duplicate row values for the group of indexed columns.

index

is a Guardian name (or an equivalent DEFINE) for the new index. The fully expanded index name must be unique in the network. If the index is partitioned, *index* identifies the primary partition.

If SMF is installed on your node, the volume portion of *index* can be either a direct or virtual volume. If you specify only a subvolume and index name, SQL creates an index in the current default volume. If the default volume is virtual, the index resides on the virtual volume. If the default volume is direct, the index resides on the physical volume as a direct file not managed by SMF.

index can reside on any node or volume, independent of the location of the underlying table, but the volume on which the index is created must be audited by the TMF subsystem, even if the index itself is nonaudited. (An index is nonaudited if its underlying table is nonaudited.)

table

is the name of the table for which to create the index (or an equivalent DEFINE).

col [ASC[ENDING] | DESC[ENDING]] [*collate-spec*]

specifies a column to include in the index, the order in which to store and retrieve key values in the column within the index, and a collating sequence for the column within the index.

The number of columns allowed in an index depends on the length of the index key. For more information on limitations, see [Index Keys](#) on page I-9.

col must be a column in *table*, but does not need to be adjacent to other columns specified for the index or in the same order relative to other columns as in the table.

ASCENDING is the default order for *col*.

COLLATE { *collation* | CHARACTER SET }

specifies an alternate collating sequence for the column within the index. You can use this clause only if the associated column is of a data type that allows a collating sequence as part of its definition.

collation is the name of an existing collation (or an equivalent DEFINE) that specifies a collating sequence and uses the same character set as the associated column

CHARACTER SET specifies a collating sequence based on the binary value of characters in the column

Specifying a *collation* for the index might affect the performance of certain queries using the index because SQL cannot perform hash joins or hash groupings on indexes with collations.

The default collating sequence for the column within the index is the same as the collating sequence for the corresponding column of the underlying table.

CATALOG *catalog*

specifies the name of the catalog in which to describe the index (or an equivalent DEFINE). *catalog* is the name of the subvolume that contains the catalog. The index and catalog must be on the same node. The default is the current default catalog.

PHYSVOL *volume-name*

If SMF is installed on your node, the PHYSVOL option directs SQL to override SMF and place the index or primary partition on the physical volume *volume-name*. For *volume-name*, specify either a physical volume or equivalent DEFINE. Do not include the node name in your volume name.

You can specify a physical volume for each secondary partition in the PARTITION clause.

This option is available only if you specify a virtual volume for *index*. *volume-name* must belong to the virtual volume you specify.

INVALIDATE | NO INVALIDATE

specifies whether to invalidate programs that use the underlying table,:

INVALIDATE Invalidate all programs that use the underlying table and modify the table's redefinition timestamp

NO INVALIDATE Do not invalidate programs or modify the table's redefinition timestamp

If you do not specify either INVALIDATE or NO INVALIDATE and the underlying table has similarity checks enabled, SQL handles programs compiled with CHECK INOPERABLE plans as if you specified NO INVALIDATE and handles other programs as if you specified INVALIDATE.

In all other cases, the default is INVALIDATE.

KEYTAG *key-specifier*

specifies a two-byte key specifier unique among indexes for the table that is stored in every row of the index.

If you omit the KEYTAG clause, SQL generates a keytag for the table. System-generated keytags are sequential numbers, beginning with one. User-specified keytag values can be either two bytes of character data or a SMALLINT UNSIGNED value in the range 1 through 65535.

PARALLEL EXECUTION { ON [CONFIG *file*] | OFF }

specifies whether to load partitions of a partitioned index in parallel. (The PARALLEL EXECUTION clause has no effect when you create an index on an empty table.)

PARALLEL EXECUTION ON directs SQL to load index partitions in parallel. PARALLEL EXECUTION OFF, the default, directs SQL to load index partitions serially.

file is the name of an EDIT file (or an equivalent DEFINE) that contains instructions for configuring the processes that load the index. For more information about how to specify configuration instructions in *file*, see [Parallel Index Loading](#) on page P-5.

If *table* and *index* are not partitioned, sorting by subsort process is usually faster than the PARALLEL EXECUTION option. (Sorting by subsorts is not recommended if the index is partitioned and parallel processing is used.) You configure subsorts with class SUBSORT DEFINEs and the SUBSORT attribute of the =_SORT_DEFAULTS DEFINE. For more information on configuring subsorts, see the *FastSort Manual*. See [Examples—CREATE INDEX](#) on page C-150.

PARTITION (*partition* [, *partition*] ...)

defines secondary partitions for a partitioned index.

partition is the definition of a single secondary partition and includes the location of the partition, the first key value for the partition, and (optionally) the catalog, physical volume, and EXTENT, MAXEXTENT, and FORMAT values for the partition. For information on this clause, see [PARTITION Clause](#) on page P-16.

```

[ NAME operation-name ]
[
WITH SHARED ACCESS [ REPORT [ TO collector | ON | OFF ] ]
[
[ { COMMIT [WORK] commit-options } ]
[ { ROLLBACK [WORK] } ]

```

specifies that the table being indexed be available for read and write access by DML statements and read access by utilities throughout most of the create index operation.

The optional clauses allow you to name the operation, control EMS reporting for the operation, specify a time window for the beginning of the commit phase of the operation (the phase in which DML and utilities operations on the table are temporarily restricted), and specify the timeout period for lock requests and the handling of retryable errors during the commit phase of the operation.

You can use WITH SHARED ACCESS only if the table being indexed is audited and if each partition of the table resides on a node running version 315 or later of NonStop SQL/MP. You cannot use WITH SHARED ACCESS on a CREATE INDEX statement that executes within a user-defined transaction.

For information about operations that use WITH SHARED ACCESS, see [WITH SHARED ACCESS OPTION](#) on page W-4. For information about the optional clauses, see [NAME Option](#) on page N-2, [REPORT Option](#) on page R-3, or [COMMIT Option](#) on page C-52.

attribute-spec

specifies file attributes for the key-sequenced file that holds the index. This is a summary of the file attributes you can specify:

Attribute	Action
ALLOCATE	Controls amount of disk space allocated. Default is to allocate space as needed.
AUDITCOMPRESS	Controls whether unchanged columns are included in audit records. Default is to include only changed columns.
BLOCKSIZE	Sets size of data blocks. Default is 4096.
BUFFERED*	Turns buffering on or off.
CLEARONPURGE*	Controls disk erasure when file is dropped.
DCOMPRESS*	Controls key compression in data blocks.
DSLACK	Sets percent of slack in data blocks. Default is value of the SLACK attribute.
EXTENT	Sets extent sizes. Default is 16 pages for the first extent, 64 for others.
FORMAT*	Sets format for primary partition of the index. Only valid on a node running version 350 or later.

ICOMPRESS*	Controls key compression in index blocks.
ISLACK	Sets percent of slack in index blocks. Default is value of the SLACK attribute.
LOCKLENGTH	Sets number of leading bytes in the key to use for generic locks. Default is 0, which specifies the entire key.
MAXEXTENTS	Sets maximum extents. Default is 160.
NOPURGEUNTIL	Sets date after which drop is allowed. Default allows immediate drop.
SERIALWRITES*	Specifies serial or parallel writes.
SLACK	Sets percent of slack in blocks if not specified by DSLACK or ISLACK. Default is 15 percent.
TABLECODE*	Sets tablecode. Default is 0.
VERIFIEDWRITES*	Controls verification of writes to disk.

Attributes marked with an asterisk (*) default to the same value as the corresponding attribute in the underlying table. For information, see the entry for a specific attribute.

Considerations—CREATE INDEX

- Note that CREATE INDEX effectively invalidates online dumps of the table underlying the new index. To ensure TMF file-recovery protection, make new online dumps of all partitions of the table and its indexes. For information about online dumps, see the *SQL/MP Installation and Management Guide*.
- To create an index, you must be a generalized owner of the underlying table. You must also have authority to read and write to the underlying table, authority to write to the USAGES table of catalogs that describe the table, and authority to write to catalogs that receive the description of the index and partitions of a partitioned index, and catalogs of dependent programs.

The underlying table and any protection views declared on that table must be accessible at the time an index is created. If the table is partitioned, all partitions must be accessible.

If you omit WITH SHARED ACCESS, CREATE INDEX locks out write operations (including INSERT, DELETE, and UPDATE operations) on the table being indexed throughout the operation. If other processes have rows in the table locked when the operation begins, CREATE INDEX waits until its lock request is granted or timeout occurs. If other processes are performing cursor SELECT or set-oriented INSERT, UPDATE, or DELETE operations on the table and the CREATE INDEX statement does not specify NO INVALIDATE, CREATE INDEX preempts those processes to acquire its own lock, causing error 60 or error 8204 in the preempted processes. While the index is being created, other processes can execute SELECT statements and read-only utility operations on the table, except during the final phase of the operation, when no access by other processes is allowed.

If you specify `WITH SHARED ACCESS`, `CREATE INDEX` does not lock out `INSERT`, `DELETE`, and `UPDATE` operations to the table being indexed except for a relatively brief period during the final phase. `CREATE INDEX` does not preempt other processes to acquire its lock even then. In addition, `WITH SHARED ACCESS` includes a `COMMIT` option that allows you to control when the operation starts the commit phase and whether to retry errors such as time outs during lock requests.

You cannot perform other DDL operations on the table being indexed until the `CREATE INDEX` operation finishes, with or without `WITH SHARED ACCESS`.

An index inherits the `OWNER`, `SECURE`, and `AUDIT` file attributes from its underlying table. The security of the underlying table must authorize network access if the index is to be partitioned across nodes or if the index is created on a node different from the node on which the table resides.

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 execute a `CREATE INDEX` statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. (For information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.)

- `CREATE INDEX` operations that use `WITH SHARED ACCESS` generally take longer to complete than those that do not. However, because `WITH SHARED ACCESS` operations allow concurrent read and write access to the source partition, they cause far less application downtime than equivalent operations without `WITH SHARED ACCESS`.

The duration of a `WITH SHARED ACCESS` operation increases with the number and length of transactions on the node that contains the source partition, particularly with the number and length of transactions that involve the source partition and the amount of activity on the audit trail used for the source partition.

- If a `CREATE INDEX` operation terminates abnormally, you (or another user with access to the super ID) must remove the new index with `CLEANUP`. If the operation used the `WITH SHARED ACCESS` option to index a table with the `AUDITCOMPRESS` file attribute (the default), you must also use `ALTER TABLE` to reset the attribute.

When you create an index with a large number of partitions, the `PARTNS` catalog table and associated `IXPART01` index might become full. To correct the situation, distribute object and partition definitions across multiple catalogs. For more information about partition limits, see [Limits](#) on page L-6.

- An operation that uses `WITH SHARED ACCESS` cannot complete successfully unless the TMF audit trail generated during the operation is available for reading later in the operation. If a required audit trail has been overwritten, a `WITH SHARED ACCESS` operation cancels changes made to the database and terminates.

When performed on a base table whose partitions have valid TMF online dumps, a CREATE INDEX operation that uses WITH SHARED ACCESS generates audit information for each of the new index partitions. The index partitions might not audit to the same audit trail as the source.

In addition, a CREATE INDEX operation that uses WITH SHARED ACCESS turns off the AUDITCOMPRESS file attribute for the table being indexed for the duration of the operation. This increases the amount of audit information for the table during this period.

Lengthy operations that use WITH SHARED ACCESS might require an operator to mount tapes of previously taken TMF audit dumps. (Requests to mount TMF audit dump tapes for WITH SHARED ACCESS operations are not distinguishable from other requests to mount TMF audit dump tapes. Such requests are generally sent to an operator's console. SQL does not return information about such requests to the terminal or process that started the operation.)

- Rows in an index are stored in ascending or descending order, as defined with CREATE INDEX, for the first column and subsequent columns of the index. For sorting purposes, null values are considered greater than all other values.

If multiple index rows share the same value for the first column, SQL uses values in the second column to order the rows, and so forth. If duplicate index rows occur in a nonunique index, SQL stores duplicate index key values in ascending or descending order, depending on the sequence specified for the columns of the primary key of the underlying table.

- There is a limit on the number of indexes that can exist for a table and on the number of partitions that can exist for an index. For information, see [Limits](#) on page L-6.

The type of partition array associated with an index affects these limits. The partition array for an index is always the same type as that associated with the underlying base table.

- These considerations apply to index partition formats:
 - The default partition format is based on the partition array value of the underlying table. If the partition array of the underlying table is STANDARD or EXTENDED, the default partition format is 1. If the partition array of the underlying table is FORMAT2ENABLED, the default partition format is 2. For relative and entry-sequenced tables, the partition format is always 1 because such tables cannot have a FORMAT2ENABLED partition array.
 - An index inherits its partition array value from the underlying table. Because there is no command to explicitly set the partition array value for an index, a table must be Format 2-enabled before any of its index partitions can be Format 2.
 - When you create a table with a FORMAT2ENABLED partition array, the table will have an object version of at least version 350. Therefore, all partitions of

the table and its index's partitions must be cataloged in version 350 or later catalogs for the CREATE INDEX catalog to succeed.

Examples—CREATE INDEX

- This example creates an index on the LAST_NAME and FIRST_NAME columns of table EMPLOYEE:

```
CREATE INDEX \SYS1.$VOL1.PERSNL.EMPLOYEE0 ON
  \SYS1.$VOL1.PERSNL.EMPLOYEE (LAST_NAME, FIRST_NAME) CATALOG
  \SYS1.$VOL1.PERSNL;
```

- This example creates an index on a single column of the EMPLOYEE table and specifies a maximum number of extents for the index.

```
CREATE INDEX EMPLOYEE2 ON EMPLOYEE (JOBCODE)
  CATALOG PERSNL MAXEXTENTS 200
  WITH SHARED ACCESS NAME CR_IND_EMP2 COMMIT BY REQUEST;
...
CONTINUE CR_IND_EMP2 ONCOMMITERROR COMMIT BY REQUEST;
```

The WITH SHARED ACCESS option allows other processes to select, delete, insert, and update records in the EMPLOYEE table during most of the operation; without WITH SHARED ACCESS, other processes would be able to select from the EMPLOYEE table only during the operation. The COMMIT BY REQUEST option allows the user to control entry to the commit phase of the operation, which does lock out other processes. The CONTINUE statement starts the commit phase, directing SQL to return control to the user if a retryable error occurs during the phase.

- This example improves the efficiency of queries on customers placing orders by adding an index on the CUSTNUM column in the ORDERS table. It specifies WITH SHARED ACCESS so that—as in the previous example—there is no application downtime during most of the operation.

```
CREATE INDEX SALES.XORDCUS ON SALES.ORDERS (CUSTNUM)
  CATALOG SALES
  MAXEXTENTS 500
  WITH SHARED ACCESS REPORT ON COMMIT BY REQUEST;
...
CONTINUE CREATE_INDEX;
```

Because the statement does not specify the NAME option of WITH SHARED ACCESS, the operation name defaults to CREATE_INDEX. REPORT ON turns on EMS reporting for the WITH SHARED ACCESS operation. COMMIT BY REQUEST allows the user to control entry to the final phase of the operation, as explained in the preceding example, but in this case, the CONTINUE statement that starts the commit phase does not request user control if retryable errors occur. Any errors in the final phase of the operation cause the entire operation to be rolled back.

- This example prevents the addition of duplicate employee names to the employee table by creating a unique index on the LAST_NAME and FIRST_NAME columns:

```
CREATE UNIQUE INDEX PERSNL.XEMPNAM
  ON PERSNL.EMPLOYEE (LAST_NAME FIRST_NAME)
  CATALOG PERSNL;
```

- This example creates a nonpartitioned index using a simple parallel sort with suborts. The DEFINES set up a sort operation with four subort processes. The CREATE INDEX statement starts a sort process that uses the specified subort processes:

```
>> ADD DEFINE =_SORT_DEFAULTS, CLASS SORT, SUBSORTS (=SS1,
+> =SS2, =SS3, =SS4);
>> ADD DEFINE =SS1, CLASS SUBSORT, SCRATCH $VOL1;
>> ADD DEFINE =SS2, CLASS SUBSORT, SCRATCH $VOL2;
>> ADD DEFINE =SS3, CLASS SUBSORT, SCRATCH $VOL3;
>> ADD DEFINE =SS4, CLASS SUBSORT, SCRATCH $VOL4;
>> CREATE INDEX AGEINDEX ON CUSTABLE (COL2);
```

- This example creates a Format 2-enabled index with four Format 2 partitions (assuming that the EMPLOYEE table is a format 2-enabled table):

```
CREATE INDEX $VOL1.PERSNL.EMPLOYEE0 ON
  \SYS1.$VOL1.PERSNL.EMPLOYEE (LAST_NAME)
PARTITION ($VOL2.PERSNL.EMPLOYEE0 FIRST KEY "D" ,
           ($VOL3.PERSNL.EMPLOYEE0 FIRST KEY "K" ,
           ($VOL4.PERSNL.EMPLOYEE0 FIRST KEY "S" )
FORMAT 2;
```

CREATE SYSTEM CATALOG Command

CREATE SYSTEM CATALOG is an SQLCI command that allows the local super ID to create the system catalog, including the SQL.CATALOGS table, when NonStop SQL/MP is first installed on a node.

```
CREATE SYSTEM CATALOG [ [ catalog-name ]
                        [ PHYSVOL volume-name ] ];
```

catalog-name

specifies the location of the system catalog on the local node.

catalog-name is the volume and subvolume name of the new system catalog (or an equivalent CLASS CATALOG DEFINE). The volume you specify must be audited. The subvolume name must be unique among catalog names on the volume.

The default *catalog-name* is \$SYSTEM.SQL.

If SMF is installed on your node, *catalog-name* can be either a direct or virtual volume.

If you specify only a subvolume for *catalog-name*, SQL creates the set of system catalog tables in the current default volume. If the default is a virtual volume, the set of system catalog tables resides on the virtual volume. If the default is a physical volume, the set of catalog tables resides on the physical volume as direct files not managed by SMF.

PHYSVOL *volume-name*

If SMF is installed on your node, the PHYSVOL option directs SQL to override SMF and place the system catalog on the physical volume *volume-name*. For *volume-name*, specify either a physical volume or equivalent DEFINE. Do not include the node name in your volume name.

This option is available only if you specify a virtual volume for *catalog-name*. *volume-name* must belong to the virtual volume you specify.

Considerations—CREATE SYSTEM CATALOG

- Only the local super ID can create a system catalog.
- The TMF subsystem must be operating when you execute CREATE SYSTEM CATALOG.
- SQL creates the CATALOGS table on a subvolume named SQL on the same volume as the remainder of the system catalog. The CATALOGS table is described in the system catalog, and the system catalog is registered in the CATALOGS table.

- The security defined for the CATALOGS table is the default security for the super ID. Use ALTER TABLE to alter the security as needed to grant other users authority to read and write to the table to create catalogs.
- If you execute CREATE SYSTEM CATALOG when a system catalog already exists on the node, SQLCI reports an error.
- If SMF is installed on your node, there are two ways to place the set of catalog tables on a single physical volume:
 - Specify a direct volume and subvolume for *catalog-name*
 - Specify a virtual volume and subvolume for *catalog-name* and a physical volume that belongs to the virtual volume in PHYSVOL. Do not include the node name in your volume name.

If you specify a virtual volume for *catalog* and omit the PHYSVOL option, SQL can distribute catalog tables among multiple physical volumes in the virtual volume.

Examples—CREATE SYSTEM CATALOG

- This example creates a system catalog on \$SYSTEM.SQL:


```
CREATE SYSTEM CATALOG;
```
- This example creates a system catalog \$VOL.SUBVOL. The CATALOGS table resides on \$VOL.SQL, but the other catalog tables and indexes reside on \$VOL.SUBVOL:


```
CREATE SYSTEM CATALOG $VOL.SUBVOL;
```

CREATE TABLE Statement

CREATE TABLE is a DDL statement that creates a table.

CREATE TABLE requires you to specify a table name and a description of each column in the table, but allows you to specify many other attributes of the table as well. A typical table definition also includes a description of the primary key or clustering key for the table (which affects data retrieval and storage for the table) and the name of the catalog to receive the description of the table.

```
CREATE TABLE table { like-spec
                      { definition-spec }

[ CATALOG catalog ]
[ PHYSVOL volume-name ]
[ CLUSTERING KEY key-column-list ]
[ { ORGANIZATION } { K[EY SEQUENCED] } ]
[ { ORGANISATION } { E[NTRY SEQUENCED] } ]
[ { R[ELATIVE] } ]
[ PARTITION ( partition [ , partition ] ... ) ]
[ PARTITION ARRAY { STANDARD } ]
[ { EXTENDED } ]
[ { FORMAT2ENABLED } ]
[ SECURE "rwep" ]
[ SIMILARITY CHECK { ENABLE | DISABLE } ]
[ attribute-spec ]
```

like-spec is:

```
LIKE source-table [ WITH COMMENTS ]
                  [ WITH CONSTRAINTS ]
                  [ WITH HEADINGS ]
                  [ WITH HELP TEXT ]
```

definition-spec is:

```
{ ( col-def [ , col-def ] ...
  [ , [ PRIMARY ] KEY key-column-list ]
[ , col-def ] ... )
{ [ PRIMARY ] KEY key-column-list , col-def
[ , col-def ] ... )
```

col-def is:

```
column-name data-type
[ DEFAULT default | NO DEFAULT ] [ NOT NULL ]
[ HEADING string | NO HEADING ]
```

key-column-list for CLUSTERING KEY or PRIMARY KEY is:

```
{ col-name [ ASC[ENDING] | DESC[ENDING] ]
{ col-name [ ASC[ENDING] | DESC[ENDING]
[ , col-name [ASC[ENDING] | DESC[ENDING] ] ... ) }
```

attribute-spec is:

```

{
  ALLOCATE integer
  { AUDIT | NO AUDIT }
  { AUDITCOMPRESS | NO AUDITCOMPRESS }
  BLOCKSIZE integer
  { BUFFERED | NO BUFFERED }
  { CLEARONPURGE | NO CLEARONPURGE }
  { DCOMPRESS { 1 | 2 } | NO DCOMPRESS }
  EXTENT { ( pri-ext-size[,sec-ext-size] ) }
          { ext-size }
  FORMAT { 1|2 }
  { ICOMPRESS | NO ICOMPRESS }
  LOCKLENGTH integer
  MAXEXTENTS integer
  NOPURGEUNTIL date
  RECLENGTH integer
  { SERIALWRITES | NO SERIALWRITES }
  TABLECODE integer
  { VERIFIEDWRITES | NO VERIFIEDWRITES }
}

```

table

is a Guardian name (or an equivalent DEFINE) for the new table. The fully expanded table name must be unique in the network. If the table is partitioned, *table* identifies the primary partition.

The volume on which the table is created must be audited by the TMF subsystem, even if the table itself is nonaudited.

If SMF is installed on your node, the volume portion of *table* can be either a virtual or direct volume. If you specify only a subvolume for *table*, SQL places the table or primary table partition on the current default volume. If the default is a virtual volume, the table or primary partition resides on the virtual volume. If the default is a direct volume, the table or primary table partition resides on the physical volume as a direct file not managed by SMF.

LIKE *source-table*

directs SQL to create a table like the existing table or partition *source-table*, omitting comments, constraints, headings, and help text unless these clauses are specified:

WITH COMMENTS	uses comments from <i>source-table</i>
WITH CONSTRAINTS	uses constraints from <i>source-table</i>
WITH HEADINGS	uses headings from <i>source-table</i>
WITH HELP TEXT	uses help text from <i>source-table</i>

If you specify LIKE, you cannot specify ORGANIZATION or CLUSTERING KEY because they are defined by *source-table*.

You can use the CREATE TABLE command with the LIKE option to create a table with a similar schema as a partition of an existing table. You can also use the PARTITION clause with LIKE to create a partitioned table where all partitions of the new table will be like the specified partition of the original table. Each partition of the new table will be created using the same partition format as the specified partition of the original table. You cannot use the FORMAT clause to override the format.

For example, suppose that you have a two partition; Format 2 enabled table with a Format 1 primary partition and a Format 2 secondary partition. If you specify the primary partition as the template in a CREATE TABLE LIKE command, all partitions of the new table will be Format 1. If, instead, you specify the secondary partition as the template, all partitions of the new table will be Format 2. In either case, the new table will also be Format 2 enabled.

SQL does not apply partitions, views, indexes, or owner information from the source table to the created table. (The SQLCI DUP command applies partitions and owner information to a duplicate copy of a table and optionally duplicates views and indexes of the table.)

source-table is the name of an existing table (or an equivalent DEFINE).

col-def

defines a column in the table by specifying the name, data type, and (optionally) other information about the column.

The sum of the lengths of all columns for the table cannot exceed the maximum row length, which is the block size minus the header. For additional restrictions on the number of columns allowed, see [Limits](#) on page L-6 and [Data Types](#) on page D-1.

column-name

is an SQL identifier that is the name of a column. Each column name must be unique within the table and cannot be an SQL reserved word. You cannot use SYSKEY as a column name except when the table has a user-defined primary key.

data-type

specifies a data type for the column and (optionally, if the data type allows) an alternate character set or collation for the column. For information, see [Data Types](#) on page D-1.

A specific host language might not support all SQL data types. For more information about host language type compatibility, see the SQL/MP programming manual for your host language.

DEFAULT *default* | NO DEFAULT

specifies a default value for the column or specifies that the column does not have a default value. *default* can be a literal of one of the special values CURRENT, SYSTEM, or NULL. For information, see [DEFAULT Clause](#) on page D-26.

You must specify the DEFAULT clause if you specify the NOT NULL clause because the default is DEFAULT NULL.

NOT NULL

specifies that the column cannot contain any null values. SQL allows null values in a column unless you specify NOT NULL.

If you specify NOT NULL and NO DEFAULT, you must supply a value for the column in each row inserted. You cannot specify NOT NULL if you also specify DEFAULT NULL, either explicitly or by default.

HEADING *string* | NO HEADING

specifies a default heading for the column or specifies that the column has no default heading.

If you omit this clause, the default heading is the column name.

For more information, see [HEADING Clause](#) on page H-1.

[PRIMARY] KEY *key-column-list*

specifies the set of columns that make up the primary key for a key-sequenced table. Each column in the set must be a column previously defined for the table. The columns do not need to be contiguous, but their combined length cannot exceed 255 bytes.

SQL stores and retrieves rows in ascending or descending order, as specified, for the first column in the list. If multiple rows have the same value in the first column, SQL uses values in the second column to determine the order. If those are the same, SQL uses the third column, and so on.

You can specify only one primary key (or one clustering key) for any particular key-sequenced table. If you do not specify either the PRIMARY KEY clause or CLUSTERING KEY clause for a key-sequenced table, SQL adds a SYSKEY column to the table to use as the primary key.

Columns in the primary key definition cannot be updated and cannot contain null values, even if you omit the NOT NULL clause in the column definition.

For more information, see [Primary Keys](#) on page P-27, [Syskeys](#) on page S-90, or [Clustering Keys](#) on page C-28.

CATALOG *catalog*

specifies the catalog to hold the description of the table. The catalog and the table must be on the same node. The default is the current default catalog.

PHYSVOL *volume-name*

If SMF is installed on your node, the PHYSVOL option directs SQL to override SMF and place the table or primary table partition on the physical volume *volume-name*. For *volume-name*, specify either a physical volume or equivalent DEFINE. Do not specify the node name in your volume name.

This option is available only if you specify a virtual volume for *table.volume-name* must belong to the virtual volume you specify.

You can specify a physical volume for each secondary partition in the PARTITION clause.

CLUSTERING KEY *key-column-list*

specifies the set of columns that make up a clustering key for a key-sequenced table. Each column in the set must be a column previously defined for the table. The columns do not need to be contiguous, but their combined length cannot exceed 247 bytes (not including the 8-byte SYSKEY).

You can specify only one clustering key (or one primary key) for any particular key-sequenced table. If you do not specify either the PRIMARY KEY clause or CLUSTERING KEY clause for a key-sequenced table, SQL adds a SYSKEY column to the table to use as the primary key.

References to keys in other tables, or any references that require a unique key, should always use a primary key rather than a SYSKEY or clustering key.

Columns in the clustering key definition cannot be updated and cannot contain null values, even if you omit the NOT NULL clause in the column definition.

For information, see [Primary Keys](#) on page P-27, [Syskeys](#) on page S-90, or [Clustering Keys](#) on page C-28.

```
[ { ORGANIZATION } { K[KEY SEQUENCED] } ]
[ { ORGANISATION } { E[NTRY SEQUENCED] } ]
[ { R[ELATIVE] } ]
```

specifies the file organization for the physical file that holds the table. For more information, see [File Organizations](#) on page F-8. The default is KEY SEQUENCED.

PARTITION (*partition* [, *partition*] ...)

defines the secondary partitions of a partitioned table.

partition is the definition of a single secondary partition and includes the location of the partition, the first key value for the partition, and (optionally) the catalog, physical volume, and EXTENT and MAXEXTENTS values for the partition. For information, see [PARTITION Clause](#) on page P-16.

PARTITION ARRAY { STANDARD | EXTENDED | FORMAT2ENABLED }

specifies the type of partition array created for the underlying table and all associated indexes:

EXTENDED	specifies the extended partition array available for versions 320 and later of NonStop SQL/MP
STANDARD	specifies the type of array used by default by NonStop SQL/MP
FORMAT2ENABLED	specifies a Format 2 enabled table that can contain any combination of Format 1 or Format 2 partitions. FORMAT2ENABLED implies that the partition array has the same limits as a file with EXTENDED partition array. Only valid on a node running version 350 or later.

The size of the partition array affects how many partitions can be created for a table and its indexes. It also affects how many indexes can be created against the base table. An extended and Format 2 enabled partition array supports a larger number of indexes and table and index partitions.

PARTITION ARRAY applies to partitions created later for a table, even if the table is not initially partitioned. To change the setting for a table, use the ALTER TABLE command.

You can use the PARTITION ARRAY clause in SQLCI or in dynamic SQL statements. To check the value of PARTITION ARRAY, use the FILEINFO DETAIL command.

Tables and indexes using extended arrays require a version 320 or later catalog. DML and DDL statements on tables and indexes with extended arrays can be performed only from nodes running version 320 or later of NonStop SQL/MP. If these conditions are not met, SQL returns an error.

Tables and indexes using Format 2 enabled arrays require a version 350 or later catalog. DML and DDL statements on tables and indexes with Format 2 enabled arrays can be performed only from nodes running version 350 or later of NonStop SQL/MP. If these conditions are not met, SQL returns an error.

You can set the partition array value to FORMAT2ENABLED by either specifying PARTITION ARRAY FORMAT2ENABLED or by specifying FORMAT 2 for an individual partition.

If you omit this clause, the default PARTITION ARRAY type is STANDARD for all table types.

SECURE "rwep"

specifies the security for the table. For more information, see [Security](#) on page S-11. The default is the security of the user who executes the CREATE TABLE.

SIMILARITY CHECK { ENABLE | DISABLE }

authorizes or prohibits similarity checks on the table. The default is SIMILARITY CHECK DISABLE.

Tables that authorize similarity checks (SIMILARITY CHECK ENABLE) have version 310 or later. Such tables cannot be registered in older catalogs or accessed by older versions of NonStop SQL/MP.

attribute-spec

specifies physical file attributes for the file that holds the table. This list provides a brief description of each attribute and its default value:

ALLOCATE	Controls amount of disk space allocated. Default is to allocate space as needed.
AUDIT	Controls TMF auditing. Default is AUDIT.
AUDITCOMPRESS	Controls whether unchanged columns are included in audit records. Default is to include only changed columns.
BLOCKSIZE	Sets size of data blocks. Default is 4096.
BUFFERED	Turns buffering on or off. Default is on when audited, else it is off.
CLEARONPURGE	Controls disk erasure when file is dropped. Default is no erasure.
DCOMPRESS	Controls key compression in data blocks, and compression method. Default is no compression (no DCOMPRESS). Default compression method is 1, the former method.
EXTENT	Sets extent sizes. Default is 16 pages for the first extent, 64 for others.
FORMAT	Specifies the format for the base partition of the table. To specify the format for secondary partitions, you must specify FORMAT in the PARTITION clauses defining each partition. Only valid on a node running version 350 or later.
ICOMPRESS	Controls key compression in index blocks. Default is no compression.
LOCKLENGTH	Sets number of bytes in key to use for generic locks. Default is entire key.
MAXEXTENTS	Sets maximum extents. Default is 160.
NOPURGEUNTIL	Sets date after which drop is allowed. Default allows immediate drop.
RECLENGTH	Sets bytes reserved for a relative-file row. Default is total column lengths.

SERIALWRITES	Specifies serial or parallel writes. Default is serialwrites.
TABLECODE	Sets tablecode. Default is 0.
VERIFIEDWRITES	Controls verification of writes to disk. Default is no verification.

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

Considerations—CREATE TABLE

- CREATE TABLE requires authority to write to the catalogs that receive the description of the table and any partitions of the table.

The LIKE clause also requires authority to read the source table and the catalog that describes the source table.

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 execute a CREATE TABLE statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

- For partitioned tables, if SMF is installed on your node and you omit the PHYSVOL option, SQL can place all partitions on a single physical volume in the virtual volume. To distribute partitions among multiple physical volumes, do one of these:
 - create partitions in different virtual volumes
 - specify PHYSVOL for the primary and secondary partitions
 - specify direct file names for partitions
- A table definition must comply with NonStop SQL/MP limits. For information, see [Limits](#) on page L-6.

When you create a table with a large number of partitions, the PARTNS catalog table and associated IXPART01 index might become full. To correct the situation, distribute object and partition definitions across multiple catalogs. For information about partition limits, see [Limits](#) on page L-6.

- If you plan to add columns to a relative file organization, you should use RECLENGTH.
- The DCOMPRESS extension applies only to key-sequenced tables.
- These considerations apply to partition formats:
 - If you create a table with a FORMAT2ENABLED partition array, the table will have an object version of at least version 350. Therefore, all partitions of the table must be cataloged in version 350 or later catalogs for the CREATE TABLE command to succeed.

- Only key-sequenced tables can be Format 2 enabled and might have partitions that are either Format 1 or 2. Relative and entry-sequenced tables can only be Format 1 enabled and must have Format 1 partitions. If you attempt to specify FORMAT 2 or PARTITION ARRAY FORMAT2ENABLED for a relative or entry-sequenced table, you will receive an error. Specifying FORMAT 1 for relative or entry-sequenced partitions is allowed.
- The default partition format is based on the partition array value for key-sequenced tables. For STANDARD and EXTENDED, the default partition format is 1. For FORMAT2ENABLED, the default partition format is 2.
- If PARTITION ARRAY is explicitly specified, a value of STANDARD or EXTENDED requires the partition format for the base and all other partitions to be FORMAT 1.
- If the partition array parameter is not explicitly specified and if one or more partitions are explicitly specified as FORMAT, the result is as if you explicitly specified PARTITION ARRAY FORMAT2ENABLED. You are only required to specify either a single partition as FORMAT 2 or PARTITION ARRAY FORMAT2ENABLED to create a Format 2-enabled table where all partitions are Format 2.
- The partition array, format, and file parameters such as extent sizes must be consistent for the CREATE TABLE command to succeed. This is true for both explicitly specified and defaulted values.

Examples—CREATE TABLE

- This example creates a table named ORDERS on subvolume \$VOL1.SALES. The catalog also resides on subvolume \$VOL1.SALES and the primary key is the ORDERNUM column.

```
CREATE TABLE $VOL1.SALES.ORDERS (
  ORDERNUM      NUMERIC (6) UNSIGNED NO DEFAULT NOT NULL,
  ORDER_DATE    NUMERIC (6) NO DEFAULT NOT NULL,
  DELIV_DATE    NUMERIC (6) NO DEFAULT NOT NULL,
  SALESREP      NUMERIC (4) UNSIGNED NO DEFAULT NOT NULL,
  CUSTNUM       NUMERIC (4) UNSIGNED NO DEFAULT NOT NULL,
  PRIMARY KEY ORDERNUM
)
CATALOG $VOL1.SALES;
```

- This example creates a table with two columns: JOBCODE and JOBDESC. The table is key-sequenced, and the primary key is JOBCODE. JOBCODE cannot contain null values because it is used in the primary key.

```
CREATE TABLE \SYS1.$VOL1.PERSNL.JOB (
  JOBCODE       DECIMAL (4) UNSIGNED NO DEFAULT,
  JOBDESC       VARCHAR (18) NO DEFAULT,
  PRIMARY KEY JOBCODE )
CATALOG \SYS1.$VOL1.PERSNL ORGANIZATION KEY SEQUENCED;
```

- This example creates a table with column headings. The EMP table contains three columns, EMPNUM, EMPNAME, and SALARY, that are assigned the headings “Employee/Number,” “Employee Name,” and “Monthly Salary,” respectively. The primary key is EMPNUM.

```
CREATE TABLE EMP (
  EMPNUM  DEC(5)                NO DEFAULT  NOT NULL
    HEADING "Employee/Number",
  EMPNAME CHAR(30) UPSHIFT      NO DEFAULT
    HEADING "Employee Name",
  SALARY  DEC(8,2)              DEFAULT SYSTEM
    HEADING "Monthly Salary", PRIMARY KEY EMPNUM );
```

One row of EMP, with column headings, might appear as follows. The SALARY column has the default value, and the EMPNAME column is upshifted.

Employee Number	Employee Name	Monthly Salary
-----	-----	-----
62389	ANNA JONES	3882.50

- This example creates a table with a clustering key.

```
CREATE TABLE PROC.HISTORY (
  SYSTEM_ID      SMALLINT UNSIGNED,
  CPU            SMALLINT UNSIGNED,
  PIN            SMALLINT UNSIGNED,
  PROGRAM_FILE_NAME  VARCHAR (34)
)
CATALOG \SYS3.$VOL1.SYSCAT
ORGANIZATION KEY SEQUENCED
CLUSTERING KEY ( SYSTEM_ID, CPU, PIN );
```

- This example creates a table with Format 2 enabled partitions. Suppose that table ODETAIL contains about 5 million rows. Each row has 20 bytes: ORDERNUM, 6; PARTNUM, 4; UNIT_PRICE, 6; and QTY_ORDERED, 4. The PARTITION specification could describe 24 partitions:

```
CREATE TABLE \SYS1.$VOL1.SALES.ODETAIL (
  ORDERNUM  NUMERIC (6) UNSIGNED NO DEFAULT NOT NULL,
  PARTNUM   NUMERIC (4) UNSIGNED NO DEFAULT NOT NULL,
  UNIT_PRICE NUMERIC (8,2)          NO DEFAULT NOT NULL,
  QTY_ORDERED NUMERIC (5) UNSIGNED NO DEFAULT NOT NULL,
  PRIMARY KEY ( ORDERNUM , PARTNUM )
)
CATALOG \SYS1.$VOL1.SALES
ORGANIZATION KEY SEQUENCED

PARTITION (
  \SYS1.$VOL2.SALES.ODETAIL
  CATALOG \SYS1.$VOL1.SALES
  EXTENT (16368,64)
  MAXEXTENTS 919
  FORMAT 2
  FIRST KEY 030000
  ,
```

```

\SYS1.$VOL3.SALES.ODETAIL
CATALOG \SYS1.$VOL1.SALES
EXTENT (16368,64)
MAXEXTENTS 919
FIRST KEY 040000
,
... --indicates 20 more
\SYS5.$VOL23.SALES.ODETAIL --partition
CATALOG \SYS5.VOL1.SALES --specifications
EXTENT (16368,64)
MAXEXTENTS 919 FIRST KEY 980000 )
LOCKLENGTH 6
EXTENT (16368,64)
MAXEXTENTS 919
NOPURGEUNTIL OCT 31 2003, 23:59
NO AUDIT;

```

Some of the attributes specified apply to the entire table and some only to the primary partition. NOPURGEUNTIL, NO AUDIT, and LOCKLENGTH apply to all partitions. The example specifies NO AUDIT for an initial load operation, after which the attribute can be changed to AUDIT. The EXTENT and MAXEXTENTS attributes apply to the primary partition.

- This example creates a partitioned, key-sequenced table with a clustering key:

```

CREATE TABLE HD.HISTORY (
  REGION_ID          INTEGER,
  SYSTEM_ID          SMALLINT UNSIGNED,
  CPU                SMALLINT UNSIGNED,
  PIN                SMALLINT UNSIGNED,
  PROGRAM_FILE_NAME  VARCHAR (34),
  AGE                LARGEINT
)
CATALOG HC
CLUSTERING KEY (REGION_ID DESC, SYSTEM_ID)
PARTITION (
  \SA.$VOL1.HD.HISTORY FIRST KEY
    ( 1000 -- region_id
      -- system_id has low value
    )
  CATALOG \SA.$VOL1.HC,
  \NA.$VM.HD.HISTORY FIRST KEY
    ( 500 -- region_id
      -- system_id has low value
    )
  CATALOG \NA.$VM.HC
);

```

- This example creates a table with the primary key of the TIMESTAMP data type and partitions the table on a TIMESTAMP value. Another column has the TIME data type.

```

CREATE TABLE $VOL1.SUBV1.PARTTIME
( A TIMESTAMP DEFAULT CURRENT NOT NULL
, B TIME DEFAULT TIME "11:00:00" NOT NULL

```



```
, C VARCHAR(300) NO DEFAULT NOT NULL
, PRIMARY KEY A
)
PARTITION (
  $VOL2.SUBV1.PARTTIME FIRST KEY TIMESTAMP
  "1989-12-1:12:00:00.000000" );
```

- This example column definitions show various combinations of DEFAULT, NULL, and NOT NULL clauses and their effects.

Column PARTNUM can contain null values. Because no DEFAULT clause is specified, the column is initialized to a null value when a row is inserted without supplying a value for PARTNUM:

```
( ... PARTNUM NUMERIC (4) UNSIGNED ... )
```

Column DEPTNUM cannot contain null values. The user must supply a nonnull value when a row is inserted:

```
(... DEPTNUM NUMERIC (4) UNSIGNED NO DEFAULT NOT NULL ... )
```

Column ORDERNUM can contain null values. The column is initialized to a system default value when a row is inserted without supplying a value for ORDERNUM:

```
( ... ORDERNUM NUMERIC (6) UNSIGNED DEFAULT SYSTEM ... )
```

Column EMPNUM cannot contain null values. The column is initialized to a system default value when a row is inserted without supplying a value for EMPNUM:

```
(...EMPNUM NUMERIC (4) UNSIGNED DEFAULT SYSTEM NOT NULL...)
```

Column JOBCODE can contain null values. The user must supply a value for JOBCODE when a row is inserted. The value supplied could be a null value:

```
( ... JOBCODE NUMERIC (4) UNSIGNED NO DEFAULT ... )
```

- These examples create a Format 2 enabled table with four partitions, all of which will be Format 2:

```
CREATE TABLE $VOL1.SALES.ORDERS (
  ORDERNUM      PIX X(6) NO DEFAULT NOT NULL,
  ORDER_DATE    PIC X(6) NO DEFAULT NOT NULL,
  DELIV_DATE    PIC X(6) NO DEFAULT NOT NULL,
  SALESREP      PIC X(4) NO DEFAULT NOT NULL,
  CUSTNUM       PIC X(4) NO DEFAULT NOT NULL,
  PRIMARY KEY ( ORDERNUM ) )
PARTITION ($VOL2.SALES.ORDERS FIRST KEY "D",
           $VOL3.SALES.ORDERS FIRST KEY "K",
           $VOL4.SALES.ORDERS FIRST KEY "S")
PARTITION ARRAY FORMAT2ENABLED;
```

You could specify FORMAT 2 and get the same result:

```
CREATE TABLE $VOL1.SALES.ORDERS (
  ORDERNUM      PIX X(6) NO DEFAULT NOT NULL,
  ORDER_DATE    PIC X(6) NO DEFAULT NOT NULL,
  DELIV_DATE    PIC X(6) NO DEFAULT NOT NULL,
```

```

    SALESREP      PIC X(4) NO DEFAULT NOT NULL,
    CUSTNUM       PIC X(4) NO DEFAULT NOT NULL,
    PRIMARY KEY ( ORDERNUM ) )
PARTITION ($VOL2.SALES.ORDERS  FIRST KEY "D",
          $VOL3.SALES.ORDERS  FIRST KEY "K",
          $VOL4.SALES.ORDERS  FIRST KEY "S")
FORMAT 2;

```

- This example creates a Format 2 enabled table with four partitions. The first three partitions are Format 1 and the last is Format 2:

```

CREATE TABLE $VOL1.SALES.ORDERS (
    ORDERNUM      PIX X(6) NO DEFAULT NOT NULL,
    ORDER_DATE    PIC X(6) NO DEFAULT NOT NULL,
    DELIV_DATE    PIC X(6) NO DEFAULT NOT NULL,
    SALESREP      PIC X(4) NO DEFAULT NOT NULL,
    CUSTNUM       PIC X(4) NO DEFAULT NOT NULL,
    PRIMARY KEY ( ORDERNUM ) )
PARTITION ($VOL2.SALES.ORDERS  FIRST KEY "D", FORMAT 1
          $VOL3.SALES.ORDERS  FIRST KEY "K", FORMAT 1
          $VOL4.SALES.ORDERS  FIRST KEY "S",FORMAT 2)
PARTITION ARRAY FORMAT2ENABLED;

```

CREATE VIEW Statement

CREATE VIEW is a DDL statement that creates a view.

```

CREATE VIEW view [ ( new-name [ , new-name ] ... ) ]

    AS select-statement

[ | FOR PROTECTION | ]
[ | SIMILARITY CHECK { ENABLE | DISABLE } | ]
[ | CATALOG catalog-name | ]
[ | SECURE "rwep" | ]
[ | WITH CHECK OPTION | ]
[ | WITH HEADINGS | ]
[ | WITH HELP TEXT | ]

new-name is:

    new-column-name [ HEADING string | NO HEADING ]

```

view

specifies a Guardian name for the view (or an equivalent DEFINE). The fully expanded view name must be unique among object names in the network.

The volume on which the view is created must be audited by the TMF subsystem, even if the view itself is nonaudited.

```
[ ( new-name [ , new-name ] ... ) ]
```

specifies names for the columns of the view and, optionally, headings for the columns. If you do not specify this clause, columns in the view have the same names as the columns in the select list of *select-statement*.

No two columns of the view can have the same name; if a view refers to more than one table and the select list refers to columns from different tables with the same name, you must specify new names for columns that would otherwise have duplicate names.

new-name is:

```
new-column-name [ HEADING string | NO HEADING ]
```

new-column-name

is an SQL identifier that is not a reserved word and that is unique among column names for the view. Column names in the list must match one-for-one with columns in the select-list of the AS *select-statement* clause.

```
HEADING string | NO HEADING
```

specifies a default heading for the column. (For more information, see [HEADING Clause](#) on page H-1.)

```
AS select-statement
```

specifies the columns for the view and sets the selection criteria that determines the rows that make up the view.

select-statement cannot include a host variable, an INTO or ORDER BY clause, or (except in a subquery) the BROWSE, STABLE, or REPEATABLE access option.

A *select-statement* that defines a shorthand view can include subqueries, a FROM clause with multiple table references, and WHERE, GROUP BY, HAVING, and ALL clauses. However, if any column in the select list is a function or an expression, *select-statement* must include a column list. In addition, the DISTINCT clause is allowed only within a function; for example, this statement is allowed:

```
CREATE VIEW v ( c ) AS
  SELECT COUNT (DISTINCT LAST_NAME) FROM EMPLOYEE;
```

However, this statement is not allowed:

```
CREATE VIEW v ( c ) AS
  SELECT DISTINCT LAST_NAME FROM EMPLOYEE;
```

A *select-statement* that defines a protection view must also meet these requirements:

- The FROM clause can refer to one table (with a correlation name, if desired), but cannot refer to another view.
- The WHERE clause can refer only to columns in its select list.
- The select list cannot include expressions or functions and cannot refer to duplicate column names.
- The statement cannot be combined with another SELECT statement using a UNION operator and cannot include subqueries, the keyword DISTINCT, or the GROUP BY or HAVING clause.

If *select-statement* includes a UNION operator, the view cannot be updated and the view cannot participate in an inner or outer join.

If *select-statement* includes a LEFT JOIN operator, the view can be specified only on the lefthand side of the first LEFT JOIN.

If SMF is installed on your node, any table to which the view refers must have either a virtual or direct name.

FOR PROTECTION

specifies a protection view. If you omit this clause, the view is a shorthand view.

SIMILARITY CHECK { ENABLE | DISABLE }

authorizes or prohibits similarity checks on a protection view. (You cannot specify this clause unless you also specify the FOR PROTECTION clause.) The default is SIMILARITY CHECK DISABLE.

Views that authorize similarity checks (SIMILARITY CHECK ENABLE) have version 310 or later. Such views cannot be registered in catalogs with old versions or accessed by older versions of NonStop SQL/MP.

CATALOG *catalog*

specifies the catalog to hold the description of the view. *catalog* is the name of the subvolume that contains the catalog and that is on the same node as the view. For a protection view, *catalog* must be the catalog that holds the description of the underlying table. (If the table is partitioned, the protection view is partitioned, too, and each partition of the protection view is registered in the same catalog as the corresponding partition of the table.)

The default is the current default catalog.

SECURE "rwep"

defines the security assigned to the view. The default is the default security of the user whose process creates the view.

Security is interpreted differently for protection and shorthand views. For protection views, you must ensure that users who have write access also have read access.

You must also ensure that purge authority includes the users with authority to purge the underlying table. For a shorthand view, only purge authority has meaning, although you must specify a complete security string. Anyone with authority to read the underlying tables and views can also read the shorthand view.

For information, see [Security](#) on page S-11.

WITH CHECK OPTION

specifies that no row can be placed in the database through the view unless the row satisfies the view definition. WITH CHECK OPTION applies only to protection views. If you omit this option, a newly inserted row or an updated row need not satisfy the view definition, which means that such a row can be inserted in the table but it does not appear in the view.

WITH CHECK OPTION does not affect *select-statement*; rows must always satisfy the view definition in this case.

WITH HEADINGS

specifies that the heading for a view column is inherited from the underlying table or view column from which the new view column is derived. If you specify the **HEADING** or **NO HEADING** clause in *new-column-name*, no heading is inherited. A view column that is a function or an expression cannot inherit a heading.

WITH HELP TEXT

specifies that help text for a view column is inherited from an underlying base table or view. A view column that is a function or an expression cannot inherit help text.

Considerations—CREATE VIEW

- CREATE VIEW requires authority to write to the catalog that receives the view description and to the USAGES tables of catalogs describing the underlying tables and views.

To create a protection view, you must also be the owner of the underlying table. Any partitions or indexes of the table underlying the protection view must be accessible when you create the view. To specify write access for a protection view, you must have authority to write to the underlying table and all associated indexes unless you are the super ID. To specify read access for a protection view, you must have authority to read the underlying table and all associated indexes unless you are the super ID. For protection views managed by SMF, *view* must be the same type of name, virtual or direct, as the underlying table.

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 execute a CREATE VIEW statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. (For more information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.)

- The CREATE VIEW statement, including any name expansion from the use of asterisks in column, view, and table specifications, can have a maximum of 3,000 bytes.
- The data types of the columns of the view are inherited from the columns of the table or view in which they are defined.
- A view can have as many as 200 to 400 columns, depending on the size of the column definitions. The column definitions for the view must fit in a file label.
- A protection view and the underlying table must both be on the same volume and must both be described in the same catalog. If the table is partitioned, the protection view is partitioned, too, and each partition of the protection view is registered in the same catalog as the corresponding partition of the table.

You cannot insert data in a protection view unless the view includes all the columns of the underlying table that are defined with the NO DEFAULT option. In addition, if the table underlying the view is an entry-sequenced table or a key-sequenced table with a system-defined primary key, you cannot insert data in the view if the view definition includes the system-defined primary key column in the WHERE clause.

A protection view inherits the AUDIT attribute of the underlying table. The OWNER of a protection view is set to the owner of the underlying table.

The maximum number of protection views allowed on a table is approximately 180.

- A shorthand view is audited if all referenced tables and views are audited. A shorthand view is nonaudited if all referenced tables and views are nonaudited. A shorthand view has a mixed audit classification if some of the referenced tables or views are audited and others are nonaudited, or if one of the referenced views has a mixed audit classification.

The OWNER of a shorthand view is set to the process accessor ID of the creating process.

A shorthand view that uses UNION cannot participate in a join. In addition, a SELECT on such a view cannot specify a GROUP BY or HAVING clause or include an aggregate function on any view column. A shorthand view based on an inner or outer join cannot become an inner table of an outer join.

There is no limit on the number of shorthand views allowed on a table.

- A grouped view is a view defined with a SELECT that contains a GROUP BY or HAVING clause that is not in a subquery, contains an aggregate function in the select list, or contains another grouped view in the FROM clause.

A grouped view cannot be joined with any other table or view. A query on a grouped view cannot contain a GROUP BY or HAVING clause, nor can it specify an aggregate function on any columns of the grouped view.

Examples—CREATE VIEW

- This example creates a shorthand view that includes part numbers and supplier numbers for parts in which fewer than ten are in stock:

```
CREATE VIEW GETPARTS (PNUM, SNUM)
  AS SELECT P.PARTNUM, SUPPNUM FROM PARTLOC P, PARTSUPP S
  WHERE P.PARTNUM = S.PARTNUM AND QTY_ON_HAND < 10
  CATALOG $VOL1.INVENT;
```

- This example creates a protection view on the table EMPLOYEE that contains rows with employee numbers greater than 1000. The view is secured so that anyone on the network can read the view, but only a local user with super ID authority can write to it. Any member of the owner's user group can purge the view.

```
CREATE VIEW \SYS1.$VOL1.PERSNL.EMPVIEW
  AS SELECT *
  FROM \SYS1.$VOL1.PERSNL.EMPLOYEE WHERE EMPNUM > 1000
  FOR PROTECTION
  CATALOG \SYS1.$VOL1.PERSNL
  SECURE "N-NC"
  WITH CHECK OPTION;
```

- This shorthand view retrieves average salary for each department:

```
CREATE VIEW \SYS1.$VOL1.PERSNL.DAVGSAL (DNUM, AVSAL)
  AS SELECT DEPTNUM, AVG(SALARY)
  FROM EMPLOYEE GROUP BY DEPTNUM CATALOG PERSNL;
```

- This view retrieves the annual salary for each employee and assigns the heading "ANNUAL SALARY" to the ANNUAL_SALARY column. All other columns inherit headings from the underlying table, EMP.

```
CREATE VIEW EMPV
  (EMPNUM, EMPNAME,
  ANNUAL_SALARY HEADING "ANNUAL SALARY", DEPTNUM)
  AS SELECT EMPNUM, EMPNAME, SALARY * 12, DEPTNUM
  FROM EMP WITH HEADINGS;
```

- This example shows an inappropriate way to join CUSTOMER and ORDERS. The CUSTOMER table has 100 rows, and the ORDERS table has 300 rows. Because the SELECT statement that defines the view does not include a WHERE clause, each row in the CUSTOMER table is concatenated with each row in the ORDERS table, resulting in a view with 30,000 rows.

Each row has the number of columns indicated in the column list of the CREATE VIEW statement. Most rows in the view have no meaning because they are a concatenation of a customer with an unrelated order. (See the next example for a better way to join these two tables.)

```
CREATE VIEW BAD
  (C_CUSTNUM, O_CUSTNUM, CUSTNAME, STATE, ORDERNUM)
  AS SELECT C.CUSTNUM, O.CUSTNUM, CUSTNAME, STATE, ORDERNUM
  FROM SALES.CUSTOMER C, SALES.ORDERS O CATALOG SALES;
```

This statement creates a view that joins the CUSTOMER and ORDERS tables in a better way than in the previous example. The CREATE VIEW statement uses a WHERE clause to join the two tables only at rows in which CUSTNUM values are equal. The view can never contain more rows than the number of rows in the largest table.

```
CREATE VIEW GOOD  
    (C_CUSTNUM,O_CUSTNUM,CUSTNAME,STATE,ORDERNUM)  
AS SELECT C.CUSTNUM,O.CUSTNUM,CUSTNAME,STATE,ORDERNUM  
FROM SALES.CUSTOMER C, SALES.ORDERS O  
WHERE C.CUSTNUM = O.CUSTNUM CATALOG SALES;
```

For information about join methods, see [Joins](#) on page J-1 or the *SQL/MP Query Guide*.

CURRENT Function

CURRENT is a function that returns the current local date, time, or both as a value of type DATETIME.

SQL evaluates CURRENT only once in an SQL statement. If you use CURRENT more than once in the same statement, each reference returns the same value.

```
CURRENT [ [ start-date-time TO ] end-date-time ]
```

start-date-time is:

```
{
  YEAR
  MONTH
  DAY
  HOUR
  MINUTE
  SECOND
  FRACTION
}
```

end-date-time is:

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

[*start-date-time* TO] *end-date-time*

specifies the range of DATETIME fields on which CURRENT operates. The default is YEAR TO FRACTION(6).

precision

is an unsigned integer in the range 1 through 6 that specifies the number of significant digits with which the fraction of a second is expressed. The default is 6.

Example—CURRENT

If you execute an SQL statement on February 20, 2004 at 11:30 pm that contains this call to CURRENT:

```
CURRENT YEAR TO DAY
```

the function returns this value:

```
2004-02-20
```

CURRENT_TIMESTAMP Function

CURRENT_TIMESTAMP is an SQLCI function that returns a Julian timestamp in Greenwich mean time for the current date and time. The data type of the returned value is NUMERIC(18) or LARGEINT.

CURRENT_TIMESTAMP works in the SQLCI commands BREAK FOOTING, BREAK TITLE, DETAIL, EXECUTE, PAGE FOOTING, PAGE TITLE, REPORT FOOTING, REPORT TITLE, and SET PARAM. It does not work in DML statements or other SQL statements.

CURRENT_TIMESTAMP

Considerations—CURRENT_TIMESTAMP

- In reports, CURRENT_TIMESTAMP returns a new value for each detail line printed. If you want the value of a timestamp to remain constant throughout a report, use CURRENT instead of CURRENT_TIMESTAMP. SQLCI determines the value of CURRENT only once for each query. (For example, if you use CURRENT in a report title and footing, the printed timestamp value is the same in both places.) CURRENT returns a timestamp with data type DATETIME.
- If you specify CURRENT_TIMESTAMP as a parameter value in a SET PARAM or EXECUTE, it returns a timestamp for the time SET PARAM or EXECUTE executes. Note that the results depend on whether you set the parameter in SET PARAM or EXECUTE.

For example, these statements execute statement S (which uses parameter ?T) twice, using a different Julian timestamp for each execution:

```
EXECUTE S USING ?T=CURRENT_TIMESTAMP; EXECUTE S
USING ?T=CURRENT_TIMESTAMP;
```

In contrast, this statement also execute the same statement S twice, but use the same Julian timestamp (the time SET PARAM executed) for each execution:

```
SET PARAM ?T CURRENT_TIMESTAMP; EXECUTE S; EXECUTE S:
```

Example—CURRENT_TIMESTAMP

To print the current date in a detail line, include these items in the print list:

```
"Date: ", CURRENT_TIMESTAMP AS DATE "D2/M2/Y2", SPACE 5,
"Time: ", CURRENT_TIMESTAMP AS TIME "HP2:M2"
```

For example, on March 15, 2004 at 1:30 p.m., the detail line prints:

```
Date: 03/15/04           Time: 01:30 PM
```

Cursors

A cursor is a named mechanism defined by a `SELECT` statement and used in a host language program. An opened cursor can be thought of as scanning the set of records specified by the `SELECT` operation. The program processes a cursor like a sequential file, fetching rows one by one. The row being fetched is at the current position of the cursor. The program can use the current cursor position to designate a row to delete or update. A cursor name is an SQL identifier.

Operations for each cursor used must execute in this order:

1. `DECLARE` defines the cursor.
2. `OPEN` determines the result table to fill the cursor and, for audited tables or views, associates the cursor with a TMF transaction. The program must set values of host variables or parameters in the cursor definition before the `OPEN`.
3. `FETCH` fills the cursor on the first fetch and then locks rows according to the access specified on the `SELECT` statement associated with the cursor. If a sort is required, all rows in the result table might be retrieved at this time and placed in a temporary, sorted table.
4. `DELETE` or `UPDATE WHERE CURRENT` deletes or updates the row at the current position of the cursor.
5. `CLOSE` (or `FREE RESOURCES`) releases the result table established on the `OPEN`.

A loop can execute multiple sequences of operations 3 and 4. Operation 5 can be performed any time after operation 2.

If the cursor locks or updates an audited table, the `FETCH` operation and subsequent cursor operations must be within a TMF transaction.

A process that uses a cursor must have read authority for tables and protection views referred to in the `SELECT` associated with the cursor. If the cursor refers to a shorthand view, the process must have read authority for tables or protection views underlying the shorthand view. If the cursor declaration specifies `FOR UPDATE`, the process must also have write authority for the referenced table, protection view, and underlying table of a view. SQL checks authority to use a cursor when you execute an `OPEN` statement.

If you use a cursor to locate rows to delete without specifying `FOR UPDATE` in the declaration, SQL checks only the read authority when the `OPEN` executes, although the delete requires write authority. SQL checks for write authority when the `DELETE` executes. If your program is having problems contending for data access with other users, you can specify the `IN EXCLUSIVE MODE` clause on the `SELECT` statement in the cursor declaration so that SQL does not have to escalate the lock when an `UPDATE` or `DELETE` executes. If, however, your program is reading records concurrently accessed by a cursor defined with an `IN EXCLUSIVE MODE` clause, your program must wait for access.

Cursor Position

Cursor position is similar to record position in a sequential file. Operations cause the cursor to be positioned:

OPEN	Before the first row
FETCH	On the retrieved row (the current position)
DELETE	Between rows
UPDATE	No change (the current position)
CLOSE	No position

Your **SELECT** determines the order in which rows are returned through a cursor. To specify the order, include an **ORDER BY** clause; otherwise, the order is undefined.

Cursor Stability

Cursor stability guarantees that the row at the current position of the cursor cannot be modified by another transaction. SQL does not guarantee cursor stability unless you define the cursor with the **FOR UPDATE** clause or you specify the **REPEATABLE** access option.

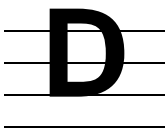
A cursor lacks stability if it points to a copy of the data and the data is concurrently available to other applications. Unless you specify the **FOR UPDATE** clause, this can happen when the **SELECT** that defines the cursor requires any of these operations:

- Ordering the rows by a column
- Removing duplicate rows
- Performing other operations requiring that the table be copied into an interim result table before use by your program

C89

The **c89** command invokes components of the C compilation system from the OSS environment. You can use it to perform any phase of a C compilation, including compiling, binding, accelerating, and SQL-compiling (compiling C programs that contain embedded SQL statements).

For information about **c89**, see the *C/C++ Programmer's Guide* or the *SQL/MP Programming Manual for C*.



Data Dictionary

The SQL/MP data dictionary is the set of all the catalogs on a network, together with the disk file labels for all the objects described in the catalogs.

A catalog is a set of tables and indexes that describes SQL/MP objects. For information, see [Catalogs](#) on page C-8.

Disk file labels are stored in directories on disk volumes. Each disk volume has a directory that contains one file label for each file on the volume. The label for a file that contains an SQL/MP object includes the name of the object, the name of the catalog that describes the object, and other information about the file. The label information enables NonStop SQL/MP to open and operate on the file without accessing the catalog.

Data Types

Each column in an SQL table is associated with a data type. You specify the data type for a column when you create the column with the CREATE TABLE or ALTER TABLE statement by using this syntax.

```
{ CHAR[ACTER] [VARYING] [(len)] [char-set] [UPSHIFT]
  [ COLLATE { collation | CHARACTER SET } ]
{
PIC[TURE] X [(len)] [DISPLAY] [ char-set ] [UPSHIFT]
  [ COLLATE { collation | CHARACTER SET } ]
{
VARCHAR[ACTER] [ ( len ) ] [ char-set ] [ UPSHIFT ]
  [ COLLATE { collation | CHARACTER SET } ]
{
NATIONAL CHAR[ACTER] [ VARYING ] [ (len) ]
{
NCHAR[ACTER] [ VARYING ] [ (len) ]
{
NUMERIC [ (digits[,scale]) ] [ SIGNED | UNSIGNED ]
{
{SMALLINT | INT[EGER] | LARGEINT } [SIGNED | UNSIGNED]
{
{FLOAT [ ( precision ) ] | REAL | DOUBLE PRECISION }
{
DEC[IMAL] ( digits[,scale] ) [ SIGNED | UNSIGNED ]
{
PIC[TURE] [S]{ 9(integer) [ V[9(scale)] ] }
              { V9(scale)
              }
  [ DISPLAY [ SIGN IS LEADING ] ]
  [ COMP ]
{ }
```

```

{ DATETIME [ start-date-time TO ] end-date-time }
{
DATE
TIME
TIMESTAMP
INTERVAL start-field [(sf-prec)] [ TO end-field ]
}

charset is:

```

```

CHAR[ACTER] SET {
ISO88591
ISO88592
ISO88593
ISO88594
ISO88595
ISO88596
ISO88597
ISO88598
ISO88599
KANJI
KSC5601
UNKNOWN
}

```

CHAR[ACTER]

specifies a column with character data.

Unless you specify additional clauses, data type CHAR specifies a fixed-length, single-byte column with an UNKNOWN character set and a collating sequence based on the binary values of the characters.

VARYING

specifies that the number of characters in a value stored in the column can be fewer than the maximum number of characters allowed in the column. Unless you specify VARYING, each item stored in the column is handled as having the maximum length for the column.

Note that values in a column declared as VARYING can be logically shorter than the maximum length, but the internal size of a VARYING column is actually two bytes larger than the size required for an equivalent column that is not VARYING.

len

is a positive integer that specifies the maximum number of characters allowed in the column.

The maximum value you can specify for *len* depends on the file organization of the table that contains the column, on whether the character set associated with

the column is a single-byte or double-byte character set, and on whether the data type declaration for the column includes the VARYING clause.

Data Type	Key- Sequenced	Relative or Entry- Sequenced
Single-byte unvarying	4061	4072
Single-byte VARYING	4059	4070
Double-byte unvarying	2030	2036
Double-byte VARYING	2029	2035

```
CHAR[ACTER] SET { ISO88591 | ISO88592 | ... UNKNOWN }
```

associates a character set with the column or indicates that the character set associated with the column is unknown.

The character set can be one of the single-byte character sets ISO 8859/1 through ISO 8859/9 or one of the double-byte character sets Kanji (KANJI) or KSC5601 (KSC5601). If you specify one of the double-byte character sets, however, the data type declaration for the item cannot include the UPSHIFT or COLLATION clause. (SQL cannot upshift double-byte characters or associate collations with double-byte characters. SQL always collates double-byte characters according to their binary values.) For information about specific character sets, see [Character Sets](#) on page C-17.

UNKNOWN specifies that the data has an unknown character set. Specifying UNKNOWN is equivalent to omitting the character set specification. SQL handles the data as 8-bit data.

UPSHIFT

directs SQL to upshift characters before storing them in the column.

You cannot specify UPSHIFT for a column associated with a double-byte character set. If you specify both the UPSHIFT clause and the COLLATE clause for a column, the rules for upshifting depend on those specified in the collation named in the COLLATE clause.

```
COLLATE { collation | CHARACTER SET }
```

specifies a collating sequence for the column. The collating sequence determines the default ordering of data returned by a SELECT and the default ordering for comparison predicates, although you can override these defaults for specific statements. For a key column, the collating sequence also determines storage order within the table.

collation is the name of an existing collation (or an equivalent DEFINE) that specifies a collating sequence that is associated with the same character set as the column

CHARACTER SET specifies a collating sequence based on the binary value of characters in the column

Specifying a *collation* for the column might effect the performance of certain queries using the column because SQL cannot perform hash joins or hash groupings on columns associated with collations.

The default is CHARACTER SET.

PIC[TURE] X [(*len*)] [DISPLAY]

specifies a column with fixed-length character data.

You can specify the number of characters in a PIC X column with either the *len* option described earlier or by specifying multiple Xs, with each X representing one character position. DISPLAY is an optional keyword that does not change the meaning of the clause.

Unless you specify additional clauses, the default for data type PIC X is a single-byte column with an UNKNOWN character set and a collating sequence based on the binary values of the characters, the same as for data type CHAR.

VARCHAR[ACTER]

specifies a column with varying-length character data. VARCHAR is equivalent to data type CHAR VARYING.

NATIONAL CHAR[ACTER]

specifies a column with double-byte character data from the system default multibyte character set.

Unless you specify additional clauses, an item in the column consists of one fixed-length, double-byte character.

NCHAR[ACTER]

is equivalent to data type NATIONAL CHARACTER.

NUMERIC [(*digits*[,*scale*)]] [SIGNED | UNSIGNED]

specifies an exact numeric column.

digits and *scale* are positive integers that specify the number of digits and the number of digits to the right of the decimal point, respectively. *digits* cannot exceed 18.

SIGNED or UNSIGNED indicates whether the column values are signed or unsigned. If *digits* is 10 or more, the values must be SIGNED.

The default is NUMERIC (1,0) SIGNED.

```
{ SMALLINT | INT[EGER] | LARGEINT } [SIGNED | UNSIGNED]
```

defines a binary integer column:

- | | |
|----------|---|
| SMALLINT | Two bytes, SIGNED or UNSIGNED; stores integers in the range unsigned 0 to 65535 or signed -32768 to +32767. |
| INTEGER | Four bytes, SIGNED or UNSIGNED; stores integers in the range unsigned 0 to 4294967295 or signed -2147483648 to 2147483647. |
| LARGEINT | Eight bytes, must be SIGNED; stores integers in the range -2^{63} to $2^{63} - 1$ (approximately 9.223 times 10 to the eighteenth power). |

The default is SIGNED.

```
{ FLOAT [ ( precision ) ] | REAL | DOUBLE PRECISION }
```

specifies a column that stores floating point values:

- | | |
|------------------|---|
| FLOAT | Stores floating point numbers in the range $\pm 8.62 \times 10^{-78}$ to $\pm 1.16 \times 10^{77}$. Uses 22 or 54 bits depending on <i>precision</i> . <i>precision</i> is an integer in the range 1 to 54. The default is 54. |
| REAL | Equivalent to FLOAT(22); stores floating point values with approximately seven decimal digits of precision. |
| DOUBLE PRECISION | Equivalent to FLOAT(54); stores floating point values with approximately 16 decimal digits of precision. |

Values stored as floating point numbers are approximate. SQL stores the values in scientific notation with a mantissa and an exponent, which decreases the precision of the stored value. Floating point data types should be used for values that are very large or very small and cannot easily be stored as one of the other numeric data types. If you can represent column values with one of the exact numeric data types (such as INTEGER or NUMERIC), choose the exact data type over the approximate data type.

```
DEC[IMAL] ( digits[,scale] ) [ SIGNED | UNSIGNED ]
```

specifies a column that stores decimal numeric values as ASCII characters.

digits and *scale* are positive integers that specify the precision in the number of digits and the number of digits to the right of the decimal point, respectively. *digits* cannot exceed 18.

SIGNED or UNSIGNED indicates whether the column values are signed or unsigned. The sign is stored as the first bit of the leftmost byte. If *digits* is 10 or more, the values must be signed.

The default is DECIMAL (1,0) SIGNED.

```
PICTURE [S]{ 9(integer) [ V[9(scale)] ] }
           { V9(scale)
           [ DISPLAY [ SIGN IS LEADING ] ]
           [ COMP      ] }
```

specifies a numeric column. If you specify COMP, the column is binary and equivalent to the data type NUMERIC. If you omit COMP, DISPLAY SIGN IS LEADING is the default and the data type is equivalent to the data type DECIMAL. The value of the number stored in the data item cannot exceed the number of 9s in the PICTURE specification.

The S specifies a signed column. The sign is stored as the first bit of the leftmost byte (digit). If you omit S, the column is unsigned.

A column with 10 or more digits must be signed.

The 9(*integer*) specifies *integer* number of digits. The value of integer must be positive.

The V designates a decimal position. The 9(*scale*) designates the number of positions to the right of the decimal point. The value of *scale* must be a positive integer. If you omit V9(*scale*), the scale is 0. If you specify V9, the scale is 1.

Instead of *integer* or *scale*, you can specify multiple 9s, with each 9 representing one digit. For example, PIC 9V999 has a scale of 3.

The values stored in the column cannot exceed a value defined by the PICTURE specification. The values of integer and scale determine the length of the column. The sum of these values cannot exceed 18.

There is no default numeric column definition. You must specify either 9(*integer*) or V9(*scale*).

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

specifies a column that contains date, time, or date and time values.

start-field and *end-field* must be one of these logically contiguous fields: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, FRACTION. For information, see [DATETIME Data Type](#) on page D-15.

DATE

is equivalent to DATETIME YEAR TO DAY.

TIME

is equivalent to DATETIME HOUR TO SECOND.

TIMESTAMP

is equivalent to DATETIME YEAR TO FRACTION(6).

INTERVAL *start-field* [(*sf-prec*)] [TO *end-field*]

specifies a column that represents a duration of time as a year-month or day-time interval. *start-field* and *end-field* specify the starting or ending field in one of these ranges of interval fields:

YEAR, MONTH

DAY, HOUR, MINUTE, SECOND, FRACTION(*fraction-prec*)

If the ending field is FRACTION, you can specify a precision of from 1 to 18 digits, within parentheses; the default is FRACTION(6).

sf-prec is an unsigned integer greater than 0 that specifies the number of significant digits allowed for the first field of INTERVAL values.

DEFAULT *default-type* | NO DEFAULT

specifies a default value for the column or specifies that the column has no default value. *default-type* must be a literal compatible with the data type of the column or one of the keywords CURRENT, SYSTEM, or NULL. For information, see [DEFAULT Clause](#) on page D-26.

The default is DEFAULT NULL.

NOT NULL

specifies that the column cannot contain any null values. For information, see [Null Values](#) on page N-7.

You cannot specify NOT NULL if you also specify DEFAULT NULL, either explicitly or by default.

DATE Data Type

An item with data type DATE represents a date according to the Gregorian calendar. Values of data type DATE are equivalent to values of data type DATETIME declared as:

```
DATETIME YEAR TO DAY.
```

For information, see [DATETIME Data Type](#) on page D-15.

Example—DATE Data Type

These examples are literals of data type DATE in (respectively) default, USA, and European format:

```
DATE "2004-01-22"
DATE "01/22/2004"
DATE "22.01.2004"
```

For information, see [Date-Time Literals](#) on page D-10.

DATE_FORMAT Option

DATE_FORMAT is an option of the SQLCI report writer SET STYLE command that specifies a default format for dates.

```
DATE_FORMAT "date-format"
```

date-format

is a string that defines a new default format for print items specified with AS DATE *. It must contain a valid numeric format as described in AS DATE/TIME.

The default format is M2/D2/Y2.

Example—DATE_FORMAT

This example sets a new default date format:

```
>> SET STYLE DATE_FORMAT "MA DB2, Y4";
```

An example of a date in this format follows:

December 25, 2004.

Date-Time Data Types

An item of a date-time data type represents a point in time. It can include a date, a time, or a date and time. There are four date-time data types:

- DATETIME
- DATE
- TIME
- TIMESTAMP

The term “date-time data type” refers to all four of these data types. The term “DATETIME data type” refers only to the first of these data types.

The DATETIME data type has a range of logically contiguous fields in this order: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION. A specific DATETIME data type has a subset or range of these fields, and a specified number of significant digits for the FRACTION field. For example:

DATETIME YEAR TO MONTH

DATETIME DAY TO FRACTION(3)

Each of the other date-time data types is equivalent to a specific range of DATETIME fields:

DATE	DATETIME YEAR TO DAY	(4 bytes)
TIME	DATETIME HOUR TO SECOND	(3 bytes)
TIMESTAMP	DATETIME YEAR TO FRACTION(6)	(11 bytes)

A specific date-time data type is compatible only with another date-time data type that has the same range of fields.

For more information about the contents of DATETIME fields, see [DATETIME Data Type](#) on page D-15.

Date-Time Functions

Date-time functions are functions that you can use in expressions that involve columns defined with the date-time data types (DATETIME, DATE, TIME, and TIMESTAMP). You can use the date-time functions anywhere an arithmetic expression is allowed.

NonStop SQL/MP provides these date-time functions:

<u>CONVERTTIMESTAMP Function</u>	Converts a Julian timestamp to a DATETIME value
<u>CURRENT Function</u>	Returns the current date, current time, or current date and time
<u>DATEFORMAT Function</u>	Formats a date-time value
<u>DAYOFWEEK Function</u>	Returns an integer representing a day of the week
<u>EXTEND Function</u>	Adjusts the range of fields for a date-time value
<u>JULIANTIMESTAMP Function</u>	Converts a date-time value to a Julian timestamp

For information, see the descriptions of specific functions.

Date-Time Literals

A date-time literal is a DATETIME, DATE, TIME, or TIMESTAMP constant that you can use in an expression, in a statement, or as a parameter value. Date-time literals have the same range of valid values as the corresponding date-time data types.

A date-time literal value can be enclosed in double quotation marks (as shown in the diagram below) or in single quotation marks, and can appear in default, USA or European format.

```

{ DATETIME "dt" [ start-field TO ] end-field }
{ DATE "date" }
{ TIME "time" }
{ TIMESTAMP "ts" }

```

dt is:

<i>default-date-time</i>	<--DEFAULT format
<i>usa-date-time</i>	<--USA format
<i>european-date-time</i>	<--EUROPEAN format
<i>default-date</i>	
<i>usa-date</i>	
<i>european-date</i>	
<i>default-time</i>	
<i>usa-time</i>	
<i>european-time</i>	

default-date-time is:

```

[[yyyy-]mm-]dd:hh[:mm[:ss[.msssss]]]

```

yyyy	Year, from 0001 to 9999
mm	Month, from 1 or 01 to 12
dd	Day, from 1 or 01 to 31
hh	Hour, from 0 or 00 to 23
mm	Minute, from 0 or 00 to 59
ss	Second, from 0 or 00 to 59
msssss	Microsecond, from 0 to 999999

usa-date-time is:

```

{mm/dd[/yyyy]}bhh[:mm[:ss[.msssss]]] [am|pm]
{[mm/]dd}

```

b	Required blank character
am	AM or am, indicating time from midnight to before noon
pm	PM or pm, indicating time from noon to before midnight
hh	0 or 00 to 23 (or to 12 with am or pm), indicating an hour

european-date-time is:

```

dd[.mm[.yyyy]]bhh[.mm[.ss[.msssss]]]

```

```

default-date is:

    { yyyy[-mm[-dd]] }      <--DEFAULT format
    { mm[-dd] }
    { dd }

usa-date is:

    { [mm/[dd/]]yyyy }     <--USA format
    { mm[/dd] }
    { dd }

european-date is:

    { [[dd.]mm.]yyyy }     <--EUROPEAN format
    { [[dd.]mm] }
    { dd }

default-time is:

    { hh[:mm[:ss[.msssss]]] } <--DEFAULT format
    { mm[:ss[.msssss]] }
    { ss[.msssss] }
    { msssss }

usa-time is:

    { hh[:mm[:ss[.msssss]]] } [ am|pm ] <--USA format
    { mm[:ss[.msssss]] }
    { ss[.msssss] }
    { msssss }

european-time is:

    { hh[.mm[.ss[.msssss]]] } <--EUROPEAN format
    { mm[.ss[.msssss]] }
    { ss[.msssss] }
    { msssss }

start-field and end-field are:

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

```

start-field must precede *end-field* and only *end-field* can use the *precision* option.


```

date is:

{ yyyy-mm-dd }
{ mm/dd/yyyy }
{ dd.mm.yyyy }

time is:

{ hh:mm:ss }
{ hh:mm:ss [ am | pm ] }
{ hh.mm.ss }

timestamp is:

{ yyyy-mm-dd:hh:mm:ss.msssss }
{ mm/dd/yyyy:hh:mm:ss.msssss [ am | pm ] }
{ dd.mm.yyyy:hh.mm.ss.msssss }

```

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

specifies a constant of data type DATETIME. The *start-field* to *end-field* clause specifies the range and precision of DATETIME fields included in the constant *dt*.

DATE *"date"* specifies a constant of data type DATE
 TIME *"time"* specifies a constant of data type TIME
 TIMESTAMP *"ts"* specifies a constant of data type TIMESTAMP

Examples—Date-Time Literals

- These examples are DATETIME literals in default, USA, and European format, respectively:

```

DATETIME "2004-01-22:13:40:05.55" YEAR TO FRACTION (2)
DATETIME "01/22/2004 01:40:05.55 PM" YEAR TO FRACTION (2)
DATETIME "22.01.2004 13.40.05.55" YEAR TO FRACTION (2)

```

- These examples are DATE literals in default, USA, and European format, respectively:

```

DATE "2004-01-22"
DATE "01/22/2004"
DATE "22.01.2004"

```

- These examples are TIME literals in default, USA, and European format, respectively:

```

TIME "13:40:05"
TIME "1:40:05 PM"
TIME "13.40.05"

```

DATEFORMAT Function

DATEFORMAT is a function that formats a date-time value in DEFAULT, USA, or EUROPEAN format. DATEFORMAT returns a value of type CHAR.

You can use DATEFORMAT wherever an arithmetic expression is allowed.

<pre>DATEFORMAT (<i>date-time-expression</i>, { DEFAULT } { USA { EUROPEAN } })</pre>

date-time-expression

is an expression that evaluates to a value of type DATETIME, DATE, TIME, or TIMESTAMP.

```
{ DEFAULT }
{ USA
{ EUROPEAN }
```

specifies a display format for a date-time value. For a description of the formats, see [Date-Time Literals](#) on page D-10.

The default is DEFAULT.

Example—DATEFORMAT

This function call converts a date-time literal in DEFAULT format to USA format:

```
DATEFORMAT( DATETIME "1989-06-20:10:20" YEAR TO MINUTE, USA )
```

It returns:

```
06/20/1989 10:20 AM
```

DATETIME Data Type

An item of data type DATETIME represents a point in time. It can include a date, a time, or a date and time.

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

start-field and *end-field* are:

```
{ YEAR }
{ MONTH }
{ DAY }
{ HOUR }
{ MINUTE }
{ SECOND }
{ FRACTION precision }
```

start-field must precede *end-field* and only *end-field* can include the *precision* option.

start-field

specifies the first field for a range of DATETIME fields.

end-field

specifies the last field for a range of DATETIME fields, or the only field for a single-field DATETIME item.

precision

is an unsigned integer in the range 1 through 6 that specifies the number of significant digits with which to express the fraction of a second for *end-field*.

Considerations—DATETIME DATA TYPE

- A specific DATETIME data type is compatible only with another DATETIME data type that has the same range of DATETIME fields, or with the equivalent DATE, TIME, or TIMESTAMP data type.

For example, these pairs are compatible data types:

```
DATETIME YEAR TO DAY and DATE
DATETIME HOUR TO SECOND and TIME
DATETIME YEAR TO FRACTION(6) and TIMESTAMP
```

These three data types are NOT compatible, although they are all DATETIME data types:

```
DATETIME YEAR TO DAY
DATETIME YEAR TO FRACTION(6)
DATETIME HOUR
```

- Range and meaning of fields within DATETIME values

Field	Range	Bytes*	Meaning
YEAR	0001 to 9999	2	Year
MONTH	1 or 01 to 12	1	Month in year
DAY	1 or 01 to 31**	1	Day in month
HOUR	0 or 00 to 23	1	Hour in day
MINUTE	0 or 00 to 59	1	Minute in hour
SECOND	0 or 00 to 59	1	Second in minute
FRACTION	0 to 999999	4	Microsecond in seconds

* Bytes is the number of bytes used to store the field in a column that includes the field. Data-time columns that allow null values are two-bytes larger than the total of the included fields.

** The DAY field is also constrained by the month and year, so the number of days in a month can never exceed the number of days in that specific calendar month.

- A DATETIME value represents a point in time according to the Gregorian calendar and a 24-hour clock in local civil time (LCT). The range of times that you can represent is:

January 1, 1 A.D., 00:00:00.000000 (low value)
 December 31, 9999, 23:59:59.999999 (high value)

(The supported range includes some dates and times that are not defined in the Gregorian calendar, such as eleven days in 1583.)

Example—DATETIME

This example creates a table with several columns that have a DATETIME data type.

```
CREATE TABLE SCHEDULE (EMPLOYEE_ID          CHAR(30),
                        LAST_SCHEDULE_CHG DATETIME YEAR TO
DAY,
                        START_WORKDAY          DATETIME HOUR,
                        END_WORKDAY            DATETIME HOUR,
                        PRIMARY KEY EMPLOYEE-ID
)
```

DAYOFWEEK Function

DAYOFWEEK is a function that reads a date-time expression and returns a type INTEGER value in the range 1 through 7 that represents the day of the week expressed by the date-time value. The value 1 represents Sunday, 2 represents Monday, and so forth.

DAYOFWEEK (*date-time-expression*)

date-time-expression

is an expression that evaluates to a value of type DATETIME, DATE, TIME, or TIMESTAMP.

Example—DAYOFWEEK

This function call returns an integer that represents the day of the week from a date-time value in the START_DATE column of a table named PROJECTS:

```
SELECT DAYOFWEEK(START_DATE) FROM PROJECTS
WHERE PROJECT_NAME = "920";
```

If the row selected looks like :

PROJECT_NAME	START_DATE	END_DATE	WAIT_TIME
920	1993-02-24:20:30	1995-03-21:20:30	20

the value returned is 1, representing Sunday.

DCL (Data Control Language) Statements

DCL (Data Control Language) is the set of SQL statements and directives that control parallel processing, name resolution, and performance-related considerations such as access paths, join methods, and locks and cursors. This table summarizes the DCL statements and directives.

DCL Statements and Directives

<u>CONTROL EXECUTOR Directive</u>	Enables or disables parallel processing of queries
<u>CONTROL QUERY Directive</u>	Specifies whether to resolve names at execution or SQL startup, whether to consider use of hash joins, and whether to optimize queries for few or many rows returned
<u>CONTROL TABLE Directive</u>	Controls locks, opens, buffers, access paths, join methods, and join sequences
<u>FREE RESOURCES Statement</u>	Closes cursors and releases locks held by a process
<u>LOCK TABLE Statement</u>	Locks a table (or the underlying tables of a view) and associated indexes
<u>UNLOCK TABLE Statement</u>	Releases locks held on nonaudited tables or views

For information, see the specific statement or directive.

DCOMPRESS File Attribute

DCOMPRESS is a Guardian file attribute that controls key compression in data blocks. DCOMPRESS applies only to key-sequenced tables and to indexes.

```
{ DCOMPRESS { 1 | 2 } | NO DCOMPRESS }
```

The table default is NO DCOMPRESS. The index default is the table value at index creation. The default compression method is 1, the former compression method.

Considerations—DCOMPRESS

- Use DCOMPRESS when you need to save disk space but do not require maximum performance. Because a given amount of disk space holds more keys if the keys are compressed, key compression reduces the use of disk space; however, it also increases overhead for accessing records.
- DCOMPRESS 2 extends the compression technique to cover non-key columns, which is not possible with DCOMPRESS 1. This strategy results in significant disk space saving at a relatively lower performance price.
- For compression method 1, the former method, a file cannot use key compression in data blocks unless its primary key meets these requirements:

- The key must begin with the first column in the table.
- The key columns must be contiguous.
- The key columns must be in ascending order.
- The key columns have only these data types:
 - Fixed-length character: CHARACTER or PIC X
 - Unsigned integer: INTEGER or SMALLINT (not LARGEINT)
 - Unsigned exact numeric: NUMERIC (1 to 9) or PIC 9V9 COMP (not NUMERIC (10) or larger or PIC S9)
 - Unsigned decimal: includes PIC 9V9 DISPLAY
- For compression method 2, the new method, these data types are supported:
 - Fixed-length character: CHARACTER or PIC X
 - Unsigned integer: INTEGER or SMALLINT (not LARGEINT)
 - Unsigned exact numeric: NUMERIC (1 to 9) or PIC 9V9 COMP (not NUMERIC (10) or larger or PIC S9)
 - Unsigned decimal: DECIMAL UNSIGNED, includes PIC 9V9 DISPLAY
 - Date and time data types: DATE, DATETIME, TIMESTAMP, and TIME
 - Multibyte character types: NATIONAL CHARACTER, NCHAR
- When a key-sequenced SQL/MP table is created with DCOMPRESS 2, an alternate method of compression is used for each data row (but the key must be a simple key). Similar to DCOMPRESS 1, the key is compressed with the leading compression. The rest of the data in each row is compressed using the SOBMAP compression, which compresses repeated data, blanks, and binary zeros.

Note. DCOMPRESS 2 is not advised when updates are routine because the cost of compression and decompression can be relatively high.

- DCOMPRESS 2 does not support these data types:
 - CHAR VARYING
 - VARCHAR
 - NUMERIC SIGNED
 - SMALLINT | INTEGER signed
 - LARGEINT signed
 - FLOAT | REAL | DOUBLE PRECISION
 - DECIMAL signed
 - INTERVAL

You cannot use DCOMPRESS on a file with a clustering key. You cannot use the COLLATE option on a key column if you will be using DCOMPRESS.

- The file system compresses keys by eliminating the leading characters duplicated from one key to the next and replacing them with a one-byte count of the duplicate characters:

Series of Uncompressed Keys	Same Keys, Compressed
JONES, JANE	0JONES, JANE
JONES, JOHN	8OHN
JONES, SAM	7SAM

Key compression can actually require an additional byte per record. It saves the most space when many key values have similar beginnings.

- Relative and entry-sequenced files always have the NO DCOMPRESS attribute, although key compression has no effect on them.

DDL (Data Definition Language) Statements

DDL (Data Definition Language) is the set of SQL statements that define, delete, or modify the SQL definition of an object or catalog. They can also change the authorization to use an object or catalog.

DDL Statements

Statement	Description (page 1 of 2)
<u>ALTER CATALOG Statement</u>	Alters security for a catalog
<u>ALTER COLLATION Statement</u>	Renames or alters security for a collation
<u>ALTER INDEX Statement</u>	Renames, adds, drops, or moves partitions, or alters security or other attributes of an index
<u>ALTER PROGRAM Statement</u>	Renames or alters security of an SQL program in a Guardian file
<u>ALTER TABLE Statement</u>	Renames, alters security of file attributes, or enables or disables similarity checks for a table; also adds columns to a table and adds, drops, or moves partitions of a table
<u>ALTER VIEW Statement</u>	Renames, alters security, or enables or disables similarity checks for a view; also adds column headings to a view
<u>COMMENT Statement</u>	Writes a comment about an SQL object to a catalog
<u>CONTINUE Statement</u>	Specifies a COMMIT option for a DDL operation ready to enter its final phase
<u>CREATE CATALOG Statement</u>	Creates a catalog

Statement	Description (page 2 of 2)
<u>CREATE COLLATION Statement</u>	Creates a collation
<u>CREATE CONSTRAINT Statement</u>	Creates a constraint
<u>CREATE INDEX Statement</u>	Creates an index
<u>CREATE TABLE Statement</u>	Creates a table
<u>CREATE VIEW Statement</u>	Creates a view
<u>DROP Statement</u>	Drops a catalog, collation, constraint, index, SQL program in a Guardian file, table, or view
<u>HELP TEXT Statement</u>	Specifies help text for a column of a table or view
<u>UPDATE STATISTICS Statement</u>	Updates statistics about the contents of a table and its indexes

For information about a specific DDL statement, see the entry for that statement.

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 execute 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.

The most common errors that occur when you attempt to execute a DDL statement on an SQL object while another DDL operation is in progress on the same object:

File-system Errors

12	File in use
40	The operation timed out
73	The table is locked
1057	Unable to access table that is being altered

SQL/MP Errors

1203	Data could not be retrieved from catalog table <i>name</i>
1222	The label of <i>name</i> could not be altered

Deadlocks

Deadlock is a block to data flow caused by processes contending for the same locked data. For example, deadlock occurs when process A locks one row and waits for another row locked by process B, while process B waits for the row locked by process A.

When deadlock causes a process to fail to acquire a requested lock within the timeout period, one of two errors occurs:

- Error 73 indicates the request was canceled successfully, and the transaction continues.
- Error 40 indicates the statement timed out after being partially executed. SQL aborts the transaction if the error occurred during an INSERT, UPDATE, or DELETE against a table with an index affected by the change; otherwise, the transaction continues. (Error 40 also occurs if a statement fails to acquire a requested lock for other reasons.)

You can control the timeout period for a table or view with `CONTROL TABLE`. You can also disable the waiting for locked data and request that immediate control be returned to SQLCI or the host program. For information, see the discussion of `TIMEOUT` and `RETURN IF LOCKED` under [CONTROL TABLE Directive](#) on page C-77.

DECIMAL_POINT Option

`DECIMAL_POINT` is an option of the SQLCI report writer `SET STYLE` command that specifies either a period or a comma as the decimal point character in numeric print items.

```
DECIMAL_POINT { "." | "," }
```

The default is ".".

Considerations—DECIMAL_POINT

- The `DECIMAL_POINT` character must be a single-byte period or comma, regardless of the character set used.
- The `DECIMAL_POINT` option does not change the decimal point character used in numeric literals or in mask display descriptors. That character (which is input to SQLCI) is always a period. The `DECIMAL_POINT` option does change the decimal point character that SQLCI prints in reports when you apply the mask to a value.

Examples—DECIMAL_POINT

- This example prints a price in European format, using an F9.2 display descriptor and a comma as the decimal character:

```
S> SET STYLE DECIMAL_POINT ", ";
S> DETAIL PRICE;
S> LIST NEXT 1;
```

```
PRICE
-----
      1300,95
```

- This example uses a period in a mask format to print a price, again using a comma as the decimal character:

```
S> SET STYLE DECIMAL_POINT ", ";
S> DETAIL PRICE AS M"9999.99";
S> LIST NEXT 1;
```

```
PRICE
-----
      1300,95
```

DECLARE CURSOR Statement

DECLARE CURSOR is a DML statement used in host programs to define a cursor and associate the cursor with a SELECT statement. The program uses the cursor to fetch rows retrieved by the SELECT statement one-by-one.

<pre>DECLARE { <i>cursor</i> CURSOR FOR <i>select-stmt</i> [FOR UPDATE OF <i>col</i> [, <i>col</i>] ...] { :<i>cursor-var</i> CURSOR FOR :<i>select-stmt-var</i> }</pre>
--

cursor CURSOR FOR *select-stmt*

specifies the cursor and the SELECT statement to associate.

cursor is an SQL identifier that is the name of the cursor and that is unique among cursor names in the program.

In static SQL, *select-stmt* is the SELECT statement itself, optionally enclosed in quotation marks. In dynamic SQL, *select-stmt* is the name of the prepared SELECT as defined in the PREPARE statement.

FOR UPDATE OF *col* [, *col*]

(static SQL only) specifies that rows selected by the cursor can be updated or deleted and identifies columns to be updated. (The columns to be updated do not need to be columns in the select list of the SELECT.)

You must use this clause if you update rows (if *stmt* includes UPDATE WHERE CURRENT OF), but it is optional if you delete rows (if *stmt* includes DELETE WHERE CURRENT OF). You cannot repeat or qualify column names, or specify SYSKEY or a user-defined primary key for a key-sequenced table.

`:cursor-var CURSOR FOR :select-stmt-var`

(dynamic SQL only) specifies host variables that contain the names of the cursor and the SELECT statement to associate.

If you use this clause, the DECLARE CURSOR statement must be in executable code (not variable declarations) and must be executed before your program references the cursor. In this case (and in no other), SQL returns information to the SQLCA and SQLSA when the DECLARE CURSOR executes.

:cursor-var is a host variable that contains the name of the cursor. The cursor name must be unique among cursor names in the program. *cursor-var* must be a fixed or variable-length string in the host language.

:select-stmt-var is a host variable that contains the name of a SELECT statement or the name of a host variable that is defined in a PREPARE statement in the current program. *select-stmt-var* must be a fixed or variable-length string in the host language.

Considerations—DECLARE CURSOR

- In static SQL, a cursor declaration must compile before other statements that reference the cursor. In dynamic SQL, a cursor declaration must execute before other statements that reference the cursor.
- A SELECT statement in a DECLARE CURSOR statement cannot include an INTO clause.

If a SELECT includes an ORDER BY clause, the ORDER BY sort specification can contain a column name or integer followed by the order-designating keyword ASC or DESC. The integer designates a position in the select list, starting from 1. The column name is not required in the sort specification.

You can also use expressions in the select list.

If a SELECT in a DECLARE CURSOR updates or deletes rows:

- The FROM clause can include only one table or protection view and cannot include a JOIN operator.
- The table referred to in the SELECT must not appear in any subquery in the WHERE clause.
- The SELECT cannot include aggregate functions, the keyword DISTINCT, a shorthand view, a union operator, or a GROUP BY, HAVING, ORDER BY, or BROWSE access clause. The only exception to this rule is: if you use a cursor to locate rows to delete without specifying the FOR UPDATE OF clause in the

cursor declaration, you can include an ORDER BY clause if you are sure that SQL will choose a plan that satisfies the specified order without sorting the rows.

- DECLARE CURSOR does not acquire locks. Locks are acquired when you execute a FETCH on the cursor or—if the SELECT requires a sort—when you open the cursor. The access option you specify in the SELECT applies to rows you access with the cursor.

If the cursor deletes rows, you must ensure that your program requests locks for the deletions by including the FOR UPDATE clause, using a LOCK TABLE statement preceding the FETCH on the cursor, or using STABLE or REPEATABLE access in the SELECT. In the latter case, the row usually is held with a shared lock that is escalated to an exclusive lock for the DELETE. If the SQL cannot obtain the exclusive lock before timeout occurs, the DELETE operation can fail.

Examples—DECLARE CURSOR

- This static SQL statement defines a cursor for a read:

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR
      SELECT COL1, COL2, COL3, COL4 FROM =PARTS
      WHERE COL2 >= :HOSTVAR2 ORDER BY COL2 BROWSE ACCESS;
```

- This static SQL statement defines a cursor for an update. The FOR UPDATE clause lists the columns to be updated.

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR
      SELECT COL1, COL2, COL3, COL4 FROM =PARTS
      WHERE (COL2 = :HOSTVAR2) STABLE ACCESS
      FOR UPDATE OF COL2, COL3, COL4;
```

- This dynamic SQL example defines a cursor for a SELECT stored in a C host variable. (The SELECT is not shown.)

Variable declarations:

```
EXEC SQL BEGIN DECLARE SECTION;
      intext char[50];
```

```
      ...
EXEC SQL END DECLARE SECTION;
```

Executable code:

```
EXEC SQL PREPARE SELECTIT FROM :intext;
```

```
      ...
EXEC SQL DECLARE GETPARTS CURSOR FOR SELECTIT;
```

DEFAULT Clause

The DEFAULT clause specifies a default value for a column (a value to use as the value of the column when a row is inserted without one). You can specify a default value for any column you define with CREATE TABLE or ALTER TABLE.

DEFAULT	{	<i>literal</i>	}
	{	CURRENT	}
	{	SYSTEM	}
	{	NULL	}

The default is NULL.

literal

is a literal of a data type compatible with the data type defined for the associated column.

For a character column, *literal* must be a string literal of no more than eight bytes or the length of the column, whichever is less. If the character column is associated with a double-byte character set, *literal* must contain an even number of bytes. SQL pads *literal* with spaces (HEX 20) when inserting the value into longer character fields. (SQL always uses HEX 20 for padding, whether a single-byte or double-byte character set is associated with the expression.)

For a numeric column, *literal* must be a numeric literal that does not exceed the defined length of the column. The number of digits to the right of the decimal point must not exceed the scale of the column and the number of digits to the left of the decimal point must not exceed the number in the length (or length minus scale, if you specified scale for the column).

For a date-time column, *literal* must be a date-time literal with a precision that matches the precision of the column.

For an INTERVAL column, *literal* must be an INTERVAL literal that has the range of INTERVAL fields defined for the column.

CURRENT

(date-time columns only) specifies that the default value for the column is the appropriate portion of the Guardian timestamp at the time of the operation that assigns a value to the column.

If more than one date-time column is assigned a CURRENT default value in the same operation, SQL uses the same timestamp as the basis of all CURRENT values assigned in the operation, regardless of how long the operation takes.

SYSTEM

specifies that the default value depends on the data type of the column:

Data Type	Default Value
Character	
Fixed-length	A string of blanks
Variable-length	A zero-length string
Date-time	Same as CURRENT option
Interval	0
Numeric	0

NULL

specifies the null value as the default. Specifying NULL as the default adds two bytes to the size of the column.

You cannot specify NULL if you also specify the NOT NULL clause in the command that creates the column.

Example—DEFAULT

This example shows a CREATE TABLE statement that uses DEFAULT clauses to specify default values for three of the columns in the table:

```
CREATE TABLE ITEMS
  (ITEM_ID      CHAR(12) NO DEFAULT,
   DESCRIPTION  CHAR(50)  DEFAULT NULL,
   NUM_ON_HAND  INTEGER   DEFAULT 0      NOT NULL,
   DATE_ADDED   DATE      DEFAULT CURRENT NOT NULL,
   PRIMARY KEY  ITEM_ID);
```

DEFINES

A DEFINE is a named set of attribute-value pairs associated with a process. You can use DEFINES to pass information to a process when you start the process. DEFINES are often used to pass information about Guardian names.

NonStop SQL/MP allows you to use DEFINE names as logical names for tables, views, indexes, partitions, catalogs, collations, or Guardian files in SQL/MP statements. When SQL compiles such statements, it replaces the DEFINE name in the statement with the Guardian name currently associated with the DEFINE.

A DEFINE name begins with an equal sign (=) followed by a letter and can contain 2 to 24 characters, including alphanumeric characters, hyphens (-), underscores (_), and circumflexes (^). Uppercase and lowercase characters are considered equivalent in DEFINE names.

These are reasons for using DEFINE names in SQL statements:

- DEFINE names are easier to understand than Guardian names.

For example, the name =CUSTOMERS is simpler than an actual file name such as \SYS1.\$VOL2.SALES.CSTMERS.

- DEFINE names provide location independence.

For example, if you code with DEFINE names, you can rename database objects, move database objects, or change the database that a program accesses without changing source code.

NonStop SQL/MP includes a set of DEFINES that specify values for SQL operations. These DEFINES start with the characters “=_” and includes:

- =_AUDSERV_XSWAP_node
- =_DEFAULTS
- =_SORT_DEFAULTS
- =_SQL_CAT_HEAP_LIMIT
- =_SQL_CMP_EQ_LIMIT
- =_SQL_CMP_EVENT
- =_SQL_CMP_EVENT_NO0
- =_SQL_CMP_NO_KS_MJOIN
- =_SQL_cmp_node
- =_SQL_EXE_USE_SQAPVOL
- =_SQL_MSG_node
- =_SQL_RECGEN_node
- =_SQL_TM_node_vol

Each of the preceding DEFINES has an entry describing it. If using SQLCI HELP to access the text, be sure to include the “=_” characters in the DEFINE name.

Use DEFINES carefully. DEFINES that identify the wrong objects can cause unexpected results. Check that the DEFINES in effect identify the objects that you want to use. Create Guardian command files or OSS shell scripts for sets of frequently used DEFINES.

Using DEFINES

DEFMODE is an attribute of a process that controls whether you can create DEFINES from the process and whether DEFINES are propagated when the process starts another process. The process can be a TACL process, an OSS shell process, an SQLCI process, or a process of your own creation. The DEFMODE attribute can be set to ON or OFF.

If DEFMODE is ON, you can create, modify, delete, propagate, and display information about DEFINES. For example, if you start an SQLCI process from a TACL process with DEFMODE ON, DEFINES set in the TACL process are propagated to the SQLCI process. Similarly, you can set DEFINES in an OSS shell process and the DEFINES are propagated to a process you start from an OSS program with embedded SQL statements. DEFMODE ON is the default. Note that for OSS processes, DEFMODE ON becomes the default after the first add_define command is issued.

If DEFMODE is OFF, DEFINES are ignored and you cannot create new DEFINES. You can still modify, delete, and display information about existing DEFINES, but such DEFINES have no effect because they are not propagated to other programs. (The `=_DEFAULTS` system DEFINE is a special DEFINE that is an exception to this rule and that is always propagated. For information, see [=_DEFAULTS DEFINE](#) on page Z-3.)

Use these commands to work with DEFINES from SQLCI. Each command is described in more detail in a separate entry.

ADD DEFINE	Adds a DEFINE
ALTER DEFINE	Changes attributes of DEFINES
DELETE DEFINE	Deletes DEFINES
INFO DEFINE	Displays DEFINE attribute values
RESET DEFINE	Restores DEFINE attributes in the working set to their initial values
SET DEFINE	Establishes values for DEFINE attributes in the working set
SET DEFMODE	Enables or disables the use of all DEFINES in the current session and enables or disables the propagation of DEFINES from the current session
SHOW DEFINE	Displays DEFINE attribute values for the working attribute set
SHOW DEFMODE	Displays the current DEFMODE setting

TACL has similar commands with the same names as the SQLCI commands just listed. The OSS shell has similar commands, `add_define`, `del_define`, `info_define`, `set_define`, and `show_define`. For information about DEFINE-related commands in TACL or the OSS shell, see the *TACL Reference Manual* or the *Open System Services Shell and Utilities Reference Manual*, respectively.

Use these system procedures to work with DEFINES from within an SQL program. For information about the procedures, see the *Guardian Procedure Calls Reference Manual* or the *Open System Services System Calls Reference Manual*.

DEFINEADD	Adds a DEFINE
CHECKDEFINE	Checkpoints a DEFINE to a backup process
DEFINDELETE	Deletes DEFINES
DEFINDELETEALL	Deletes all DEFINES except <code>=_DEFAULTS</code> from the context of the current process
DEFINEINFO	Returns DEFINE attribute values
DEFINEMODE	Enables or disables the use of DEFINES
DEFINENEXTNAME	Returns the next DEFINE name (DEFINES are stored in ascending order by name)

DEFINEREADATTR	Returns an attribute value for a DEFINE or for the working attribute set
DEFINERESTOREWORK	Restores the working attribute set from the background set
DEFINESAVEWORK	Saves the working attribute set in the background set
DEFINESETATTR	Alters the value of an attribute in the working set, or resets the attribute
DEFINESETLIKE	Sets all attributes of the working set to match those of an existing DEFINE
DEFINEVALIDATEWORK	Checks the working set for consistency and completeness

Using DEFINES From SQLCI

These rules apply to the use of DEFINES in SQLCI:

- Make sure DEFMODE is set. To inherit DEFINES from the process that starts SQLCI, such as TACL or the OSS shell, set DEFMODE ON before you start SQLCI. To avoid inheriting DEFINES, set DEFMODE OFF before you start SQLCI, then set it ON within SQLCI.

DEFINES that you create during an SQLCI session remain in effect until you alter them, delete them, or end the SQLCI session. DEFINES you inherit from another process and then modify with SQLCI commands revert to their previous attribute values (that is, the values they had when you started SQLCI) when you end the SQLCI session. Any changes you make to inherited attributes within the SQLCI session apply only until you exit SQLCI.

- Except for statements compiled with PREPARE, SQLCI resolves DEFINE names in a statement at the time you enter the statement.

If you use PREPARE to compile an SQLCI statement when a CONTROL QUERY BIND NAMES AT EXECUTION directive is not in effect, SQLCI resolves DEFINE names in the statement when you enter the PREPARE, using the DEFINE values at that time. Subsequent executions of the statement use the PREPARE-time DEFINE values.

If you use PREPARE to compile an SQLCI statement when a CONTROL QUERY BIND NAMES AT EXECUTION directive is in effect, SQLCI resolves DEFINE names in the statement when you execute the statement, using the DEFINE values at that time. (Note that CONTROL QUERY BIND NAMES AT EXECUTION TIME must be in effect at PREPARE-time to cause execution-time DEFINE resolution; whether it is in effect at execution-time makes no difference.)

Using DEFINES With SQL Programs

These rules apply to the use of DEFINES in SQL statements within programs:

- You use TACL or OSS shell commands to supply DEFINES at compilation for DEFINE names used in preprocessor or host language statements and in the SQLCOMP or c89 command. For example, if you use DEFINE names in INVOKE statements for an SQL program that is executable from Guardian, you use TACL commands to supply corresponding DEFINES at preprocessor or host language compilation.

You normally supply DEFINES at explicit SQL compilation for any DEFINE names you use in static SQL statements. SQL attempts to resolve such DEFINE names during explicit compilation, if possible (although the names might be reresolved at load or execution time according to the rules that follow).

If corresponding DEFINES do not exist, the SQL compiler issues a warning and produces a program that, although valid, requires recompilation with appropriate DEFINE values. (SQL can automatically recompile programs at run time, as discussed in the SQL/MP programming manual for your host language. If you prohibit recompilation, however—or if appropriate DEFINES are still missing by the time the statement executes—an error occurs.)

Dynamic SQL statements are not affected by explicit compilation, so there is no reason to supply DEFINES for those statements before execution.

For SQL programs executable from Guardian, you can use the EXPLAIN DEFINES option on the SQLCOMP command to automatically generate an OBEY command file that contains ADD DEFINE commands for the DEFINES used in your program. To generate the OBEY command file in TACL OBEY format, use the OBEY FORM option.

You can use the command file to add the same DEFINES again at run time. For information, see the SQL/MP programming manual for your host language.

- To use DEFINES in programs, you can inherit DEFINES from your TACL or OSS shell process or use system procedure calls to create DEFINES within your program. If a DEFINE does not exist at execution time, SQL uses the stored define that existed at compilation time.

You can direct your program to access a different set of objects than the ones you specified at compilation by supplying different DEFINE values at run time than at compile time. Depending on the compilation options you specified, the similarity between the objects, and the SIMILARITY CHECK attribute of the objects, changing DEFINE values at run time causes successful similarity checks, recompilation, or errors.

To inherit DEFINES in a program, set DEFMODE ON before you start the program; to avoid inheriting DEFINES, set DEFMODE OFF before you start the program, then set it ON within the program.

DEFINES you create from an executing program remain in effect until you alter them, delete them, or terminate the program. DEFINES you inherit from another process and then modify within an executing program revert to their previous attribute values (that is, the values they had when you started the program) when the program terminates. Any changes you make to inherited attributes within the program apply only within the program.

- DEFINE names in a static SQL statement that was compiled WITHOUT a CONTROL QUERY BIND NAMES AT EXECUTION directive in effect are resolved at SQL-load time (just before the first SQL statement in the program executes) by using the DEFINES in effect at that time. Changing DEFINE values during program execution has no effect on such a statement.

DEFINE names in a static SQL statement that was compiled with a CONTROL QUERY BIND NAMES AT EXECUTION directive in effect are resolved just before each execution of the statement. Changing DEFINE values during program execution affects such a statement.

- DEFINE names in a dynamic SQL statement that is compiled by a PREPARE operation when a dynamic CONTROL QUERY BIND NAMES AT EXECUTION directive is not in effect are resolved during the PREPARE operation. Changing DEFINE values after the PREPARE but before a corresponding EXECUTE does not affect such a statement.

DEFINE names in a dynamic SQL statement that is compiled by a PREPARE operation when a dynamic CONTROL QUERY BIND NAMES AT EXECUTION directive is in effect are resolved each time a corresponding EXECUTE occurs. Changing DEFINE values after the PREPARE but before a corresponding EXECUTE affects such a statement.

DEFINE names in a dynamic SQL statement compiled by an EXECUTE IMMEDIATE statement are resolved when the EXECUTE IMMEDIATE statement executes.

DEFINE Attributes

Each DEFINE has a set of attributes associated with it. The CLASS attribute determines the function of the DEFINE:

CATALOG	Specifies catalog redirection or substitution
DEFAULTS	Specifies process defaults such as default volume and subvolume
MAP	Specifies redirection or substitution for a table, view, index, collation, partition, or program name
SEARCH	Specifies subvolumes for resolving file names in a search list
SORT	Specifies parameters for FastSort processes

SUBSORT Specifies parameters for parallel sort processes

SPOOL Sets parameters for the spooler

TAPE Specifies the attributes of a file on a labeled tape, such as block size and density

MAP is the default.

The CATALOG, DEFAULTS, and MAP classes are particularly useful with SQL and are discussed in more detail in this section.

DEFINES of Class CATALOG

DEFINES of class CATALOG specify a logical name for a particular SQL catalog. For example, this SQLCI statement assigns the logical name =PCAT to the catalog that resides on subvolume \D.\$E.F:

```
ADD DEFINE =PCAT, CLASS CATALOG, SUBVOL \D.$E.F;
```

This statement, executed while the preceding DEFINE is in effect, creates a table on \D.\$E.F:

```
CREATE TABLE T ... CATALOG =PCAT;
```

A DEFINE of class CATALOG does not change the current default catalog. (You can change the current default catalog by using the SQLCI CATALOG command or by altering the =_DEFAULTS DEFINE.)

DEFINES of Class DEFAULT

The only DEFINE of class DEFAULT of interest to a NonStop SQL/MP user is the =_DEFAULTS DEFINE, which is a system DEFINE.

The file system uses the =_DEFAULTS DEFINE to save the names of the current default node, volume, subvolume, and catalog, so that processes can share these names. When you change your current default node, volume, subvolume, or catalog, the =_DEFAULTS DEFINE is automatically modified. The values in the =_DEFAULTS DEFINE determine how to expand partially qualified Guardian names.

You cannot delete or rename the =_DEFAULTS DEFINE, but you can display and alter the values of its attributes. For example, this ALTER DEFINE command changes the current default catalog:

```
ALTER DEFINE =_DEFAULTS, CATALOG \A.$B.C;
```

For information, see [=_DEFAULTS DEFINE](#) on page Z-3 or [System DEFINES](#) on page S-94.

DEFINES of Class MAP

A DEFINE of class MAP associates a DEFINE name with the name of a table, view, index, collation, partition, or program. You can use the DEFINE name in SQL

statements as the logical name of a table, view, index, collation, partition, or program, altering the DEFINE (but not the SQL statement) when you want to point to a different physical entity.

For example, this command adds a DEFINE that assigns the logical name =ORDERS to the table whose name is \SYS1.\$VOL2.SALES.ORDERS:

```
ADD DEFINE =ORDERS, CLASS MAP, FILE \SYS1.$VOL2.SALES.ORDERS;
```

While this DEFINE is in effect, you can refer to the table as =ORDERS in SQL statements.

MAP is the default class unless the working attribute set specifies a different class, so the previous command is normally equivalent to this one:

```
ADD DEFINE =ORDERS, FILE \SYS1.$VOL2.SALES.ORDERS;
```

The working attribute set is a set of default attribute values used when you create a new DEFINE and do not explicitly specify its attributes. For information about the working attribute set, see [SET DEFINE Command](#) on page S-32, [RESET DEFINE Command](#) on page R-12, and [SHOW DEFINE Command](#) on page S-48.

Summary of DEFINE Attributes

The other attributes of a DEFINE vary according to its class. [Table D-1](#) lists the DEFINE classes and the attributes of each class. It also supplies a brief definition of attributes commonly used with SQL. For information about attributes of other classes, see the *TACL Reference Manual*.

Table D-1. Attributes of DEFINES (by Class) (page 1 of 3)

Class	Attributes	Required/ Optional
CATALOG	SUBVOL <i>subvol</i> specifies an actual subvolume name that identifies a catalog; <i>subvol</i> is a Guardian subvolume name.	Required
DEFAULTS	CATALOG[<i>\node.</i>][\$ <i>vol.</i>] <i>subvol</i> specifies the current default catalog. If you omit <i>\node</i> or <i>\$vol</i> , the current value of the corresponding element in the =_DEFAULTS DEFINE VOLUME attribute is used.	Optional
	VOLUME[<i>\node.</i>][\$ <i>vol.</i>] <i>subvol</i> specifies the current default volume and subvolume. If you omit <i>\node</i> , the system on which you are running the process is used.	Required
MAP	FILE <i>file-name</i> specifies an actual file to use when you refer to the associated DEFINE name in a command.	Required
SEARCH	SUBVOL0 <i>subvol-name</i>	Optional
	RELSUBVOL0 <i>subvol-name</i>	Optional

Table D-1. Attributes of DEFINES (by Class) (page 2 of 3)

Class	Attributes	Required/ Optional
SORT	SUBVOL1 <i>subvol-name</i>	Optional
	RELSUBVOL1 <i>subvol-name</i>	Optional
	...	
	SUBVOL20 <i>subvol-name</i>	Optional
	RELSUBVOL20 <i>subvol-name</i>	Optional
	BLOCK <i>block-size</i>	Optional
	CPU <i>cpu-number</i>	Optional
	CPUS <i>subsort-cpu-list</i>	Optional
	MODE <i>mode-type</i>	Optional
	NOTCPUS <i>cpu-list-not-subsort</i>	Optional
	NOSCRATCHON (<i>volume-name</i> [, <i>volume-name</i>]...)	Optional
	PRI <i>process-priority</i>	Optional
	PROGRAM <i>file-name</i>	Optional
	SCRATCH <i>file-name</i>	Optional
	SCRATCHON (<i>volume-name</i> [, <i>volume-name</i>]...)	Optional
	SEGMENT <i>extended-segment-size</i>	Optional
	SUBSORTS <i>define-list</i>	Optional
	SWAP <i>file-name</i>	Optional
	VLM {ON OFF}	Optional
SPOOL	BATCHID <i>job-id</i>	Optional
	BATCHNAME <i>batch-name</i>	Optional
	COPIES <i>number</i>	Optional
	FORM <i>form-name</i>	Optional
	HOLD {ON OFF}	Optional
	HOLDAFTER {ON OFF}	Optional
	LOC <i>destination</i>	Required
	MAXPRINTLINES <i>number</i>	Optional
	MAXPRINTPAGES <i>number</i>	Optional
	OWNER <i>owner-id</i>	Optional
	PAGESIZE <i>number</i>	Optional
	REPORT <i>report-name</i>	Optional
	SELPRI <i>priority-number</i>	Optional

Table D-1. Attributes of DEFINES (by Class) (page 3 of 3)

Class	Attributes	Required/ Optional
SUBSORT	BLOCK <i>size</i>	Optional
	CPU <i>cpu-number</i>	Optional
	NOSCRATCHON (<i>volume-name</i> [, <i>volume-name</i>]...)	Optional
	PRI <i>process-priority</i>	Optional
	PROGRAM <i>file-name</i>	Optional
	SCRATCH <i>file-name</i>	Optional
	SCRATCHON (<i>volume-name</i> [, <i>volume-name</i>]...)	Optional
	SEGMENT <i>extended-segment-size</i>	Optional
	SWAP <i>file-name</i>	Optional
TAPE	BLOCKLEN <i>block-length</i>	Optional
	DENSITY {800 1600 6250}	Optional
	DEVICE \$ <i>device-name</i>	Optional
	EBCDIC {IN OUT ON OFF}	Optional
	EXPIRATION <i>date</i>	Optional
	FILEID <i>file-name</i>	Optional
	FILESECT <i>volume-order</i>	Optional
	FILESEQ <i>file-order</i>	Optional
	GEN <i>generation-number</i>	Optional
	LABELS	Optional
	{ANSI IBM OMITTED BYPASS BACKUP}	
	OWNER <i>owner-id</i>	Optional
	MOUNTMSG "text"	Optional
	RECFORM {F U}	Optional
	RECLEN <i>record-length</i>	Optional
	REELS <i>num-of-volumes</i>	Optional
	RETENTION <i>days</i>	Optional
	SYSTEM \system-name	Optional
	TAPEMODE {STARTSTOP STREAM}	Optional
	USE {IN OUT EXTEND OPENFLAG}	Optional
	VERSION <i>number</i>	Optional
	VOLUME { <i>volume-id</i> SCRATCH}	Optional

Examples—DEFINES Used With SQL Programs

- This example uses the DEFINE name =PARTS to identify a table in an INVOKE statement in a COBOL program. The DEFINE for =PARTS must exist when you run the preprocessor before the COBOL compilation.

```
EXEC SQL
  INVOKE =PARTS AS PARTS-REC LEVEL (01,04)
END-EXEC.
```

- This example uses the DEFINE name =PARTS to identify a table in an INSERT statement. The DEFINE for =PARTS must exist when you run the program and should normally exist when you compile the program.

```
EXEC SQL
  INSERT INTO =PARTS
    VALUES (:PARTNUM      OF PARTS,
            :PARTDESC      OF PARTS,
            :PRICE         OF PARTS,
            :QTY-AVAILABLE OF PARTS)
END-EXEC.
```

- This example uses the DEFINE name =SCAT in a TACL command that calls the SQL compiler. =SCAT identifies the catalog in which to register the compiled program. The DEFINE for =SCAT must be in effect when you issue the SQLCOMP command.

```
SQLCOMP /IN object-file, OUT list-file>/ CATALOG =SCAT
```

DELETE DEFINE Command

DELETE DEFINE is an SQLCI command that deletes DEFINES. (DELETE DEFINE is similar to the TACL command DELETE DEFINE and the OSS shell command del_define.)

```
DELETE DEFINE { define
                ( define [ , define ] ... ) } ;
                **
                =*
```

define

is the name of a DEFINE to delete.

** or =*

specifies that all DEFINES are to be deleted.

Examples—DELETE DEFINE

- These two SQLCI commands delete the DEFINE =CAT and delete the =EMPLOYEE, =PARTLOC, and =PARTSUPP DEFINES:

```
DELETE DEFINE =CAT;
DELETE DEFINE (=EMPLOYEE, =PARTLOC, =PARTSUPP);
```

- This SQLCI command deletes all DEFINES except =_DEFAULTS specified within the SQLCI session. After the command executes, DEFINES inherited from the TACL or the OSS shell are no longer in effect for the SQLCI session, although inherited DEFINES remain in effect for the TACL or OSS shell.

```
DELETE DEFINE =*;
```

DELETE Statement

DELETE is a DML statement that deletes rows from a table or protection view.

DELETE does not delete a table or protection view, even if you delete the last row in the table or view. Use DROP to delete a table or view.

```
DELETE FROM name

    [ [ WHERE search-cond ] [ [FOR] { STABLE          } ACCESS ] ]
    [                               [ { REPEATABLE }                ] ]
    [                               ]
    [ WHERE CURRENT OF cursor-name ]
```

name

is the name of the table or protection view (or an equivalent DEFINE) from which to delete rows.

The file organization of the table or underlying table must be key-sequenced or relative. You cannot use DELETE to delete rows from an entry-sequenced table, from a catalog table, or from a shorthand view.

WHERE *search-cond*

is a search condition that specifies criteria for selecting rows to delete. The search condition cannot include a subquery that refers to *name* or its underlying table.

If you omit both the WHERE and WHERE CURRENT clauses, SQL deletes all rows.

```
[FOR] { STABLE          } ACCESS
      { REPEATABLE }
```

determines the locking for the rows to delete.

STABLE locks all data accessed but releases locks on unmodified rows without waiting for the end of the transaction.

REPEATABLE locks all data accessed until the end of the transaction.

The default is FOR STABLE ACCESS. For information, see [Access Options](#) on page A-1.

WHERE CURRENT OF *cursor-name*

(for use in programs only) specifies the name of a cursor that is positioned at the row to delete. If *cursor-name* points to an audited table or view, the DELETE must execute within a TMF transaction that also includes the FETCH for the row.

You cannot specify WHERE CURRENT OF when using parallel execution.

Considerations—DELETE

- DELETE requires authority to read and write to the table or view being deleted and to any table or view in a subquery of the search condition.
- Rows must be locked to be deleted. The locking used depends on the access option you specify in the WHERE clause or the access option you specify in the SELECT portion of the associated DECLARE CURSOR statement.

If the deletion occurs through a search condition, the access option you specify in a subquery determines the duration of locks applied to data in the tables and views referred to in the subquery.

The default access option is STABLE.

- When using an SQL cursor in a host program, a DELETE WHERE CURRENT generally provides a performance benefit over a stand-alone DELETE. SQL uses virtual sequential block buffering (VSBB) for updates through a cursor unless you use another cursor or a stand-alone DELETE or UPDATE for the same table within the same program. (Using a stand-alone DELETE or another cursor to access the table within the same process, either directly or through a view, invalidates VSBB and can degrade performance substantially.)

SQL returns these values to SQLCODE after a DELETE:

```

0      Successful DELETE
100    No rows found for a search condition
> 0    Warning code number
< 0    Error code number

```

After a successful DELETE, the SQLCA contains the exact number of deleted rows.

Examples—DELETE

- This example on DELETE removes the row for JOHN WALKER from the EMPLOYEE table:

```

DELETE FROM PERSNL.EMPLOYEE
WHERE FIRST_NAME = "JOHN" AND LAST_NAME = "WALKER";

```

- This example on DELETE removes from the table ORDERS any orders placed with sales representative 568 by any customer except customer number 3210:

```

DELETE FROM SALES.ORDERS
WHERE SALESREP = 568 AND CUSTNUM <> 3210;

```

- This example on DELETE removes from the table PARTSUPP all suppliers who charge more than \$1600.00 for items that have part numbers in the range 6400 to 6700. The DELETE uses REPEATABLE access (which provides maximum

consistency but reduces concurrency), so it would be best to execute it at a time when only a few users need concurrent access to the database.

```
DELETE FROM INVENT.PARTSUPP
WHERE PARTNUM BETWEEN 6400 AND 6700
AND PARTCOST > 1600.00 REPEATABLE ACCESS;
```

- This example on DELETE removes all suppliers not in Texas from the table PARTSUPP:

```
DELETE FROM INVENT.PARTSUPP
WHERE SUPPNUM IN (SELECT SUPPNUM FROM INVENT.SUPPLIER
WHERE STATE <> "TEXAS");
```

You can achieve the same result with this DELETE, as long as SUPPNUM does not contain NULL characters:

```
DELETE FROM INVENT.PARTSUPP
WHERE SUPPNUM NOT IN (SELECT SUPPNUM FROM INVENT.SUPPLIER
WHERE STATE = "TEXAS");
```

DESCRIBE INPUT Statement

DESCRIBE INPUT is a dynamic SQL statement that returns descriptions of input parameters for a previously prepared statement. (An input parameter is a placeholder for a literal value to be supplied when the statement executes.)

```
DESCRIBE INPUT { stmt-name } INTO :sqlda
                { :stmt-variable }
[ NAMES INTO :names-buffer ]
```

stmt-name

is the SQL identifier of the prepared SQL statement, optionally qualified with the name of the program.

:stmt-variable

is the name of a host variable that stores the SQL identifier of the prepared SQL statement.

INTO :*sqlda*

specifies the host variable name of an SQLDA declared in an INCLUDE SQLDA statement into which DESCRIBE returns the number of input parameters and a description of each parameter. (For information about the contents and use of an SQLDA, see the SQL/MP programming manual for your host language.)

NAMES INTO :*names-buffer*

specifies the host variable name of a names buffer declared in an INCLUDE SQLDA statement or elsewhere in your program into which DESCRIBE returns the names of the input parameters.

If you specify the NAMES INTO clause, DESCRIBE INPUT sets the VAR_PTR field for each entry in the SQLDA to the address of the corresponding entry in the names buffer.

Each entry in the names buffer is in VARCHAR format: the entry begins with a 2-byte numeric prefix that contains the length of the name; the SQL identifier that is the name itself follows. (The question mark that precedes a parameter name in an SQL statement is not included in the name.) Unnamed parameters have a length of 0. If the name has an odd number of characters, it is followed by a blank to make it an even length. If a SELECT column is a constant or expression, the name entry has a length of 0. The table name could be the actual table name itself, a user specified correlation name, or a DEFINE name for the table (without the leading = sign). The precedence is: correlation name first, define name second, and the actual table name the last. For example:

Example 1:

```
STATEMENT
  SELECT C1 FROM TABLE1 WHERE C1 > 100;

NAMES-BUFFER (OUTPUT)
  {09}TABLE1.C1
```

Example 2:

```
STATEMENT
  SELECT C1 FROM TABLE1 T1 WHERE T1.C1 > 100;

NAMES-BUFFER (OUTPUT)
  {05}T1.C1
```

Example 3:

```
STATEMENT
  SELECT C1 FROM =TABLE_NAME_1 WHERE C1 > 100;

NAMES-BUFFER (OUTPUT)
  {15}TABLE_NAME_1.C1
```

Example 4:

```
STATEMENT
  SELECT C1 FROM =TABLE_NAME_1 T1 WHERE T1.C1 > 100;

NAMES-BUFFER (OUTPUT)
  {05}T1.C1
```

If your program uses indicator parameters to handle null values, DESCRIBE INPUT also returns the names of the indicator parameters to the names buffer. The

IND_PTR field of each SQLVAR entry contains the address of the corresponding indicator parameter entry.

Example—DESCRIBE INPUT

This C statement returns descriptions of input variables in the prepared statement identified by :stmt_name to an SQLDA structure pointed to by :*input_sqlda_ptr.

```
EXEC SQL DESCRIBE INPUT :stmt_name INTO :*input_sqlda_ptr
                        NAMES INTO :*input_namesbuf_ptr;
```

The NAMES INTO clause directs DESCRIBE INTO to return the names of the input variables to the buffer pointed to by :*input_namesbuf_ptr and to set the var_ptr field of each entry in the SQLDA to the address of the corresponding name.

For examples of related statements and a detailed discussion of dynamic SQL programming techniques, see the *SQL/MP Programming Manual for C* or the *SQL/MP Programming Manual for COBOL*.

DESCRIBE Statement

DESCRIBE is a dynamic SQL statement that returns descriptions of output variables (usually SELECT columns) from a previously prepared statement.

```
DESCRIBE { stmt-name } INTO :sqlda
        { :stmt-variable }

[ NAMES INTO :names-buffer ]

[ COLLATIONS INTO :collations-buffer ]
```

stmt-name

is the SQL identifier of the prepared SQL statement.

:stmt-variable

is the name of a host variable that stores the SQL identifier of the prepared SQL statement.

INTO :sqlda

specifies the host variable name of an SQLDA declared in an INCLUDE SQLDA statement into which DESCRIBE returns:

For a SELECT	The number of SELECT columns and a description of each column
For an INSERT	The data type, length, and scale for the system-defined primary key (SYSKEY) of the last record inserted

If the SQLDA is not large enough to describe all the SELECT columns, DESCRIBE returns only the descriptions of the first n columns in the SELECT, where n is the number of entries in the SQLDA. (In either case, DESCRIBE does not modify the NUM_ENTRIES field in the SQLDA, which indicates the number of entries the SQLDA can hold.)

For information about the contents and use of an SQLDA, see the SQL/MP programming manual for your host language.

NAMES INTO :*names-buffer*

specifies the host variable name of a names buffer declared in an INCLUDE SQLDA statement or elsewhere in your program into which DESCRIBE returns the names of the SELECT columns (or the SYSKEY column) described in the SQLDA.

If you specify the NAMES INTO clause, DESCRIBE sets the VAR_PTR item for each entry in the SQLDA to the address of the corresponding entry in the names buffer. (If the buffer is not large enough to contain all the names, DESCRIBE sets the VAR_PTR field for any name that does not fit to a value less than 0; if the buffer is larger than necessary, DESCRIBE ignores the extra bytes.)

Each entry in the names buffer is in VARCHAR format: the entry begins with a 2-byte NUMERIC prefix that contains the length of the name and the name follows, in the form:

tablename.columnname

If the name has an odd number of characters, it is followed by a blank to make it an even length. If a SELECT column is a constant or expression, the name entry has a length of 0.

A SYSKEY column name for a table is *table-name.SYSKEY*. A SYSKEY column name for a view is *view-name.derived-column-name*.

The table name could be the actual table name itself, a user specified correlation name, or a DEFINE name for the table (without the leading = sign). The precedence is: correlation name first, define name second, and the actual table name the last. For example:

Example 1:

```
STATEMENT
  SELECT C1 FROM TABLE1 WHERE C1 > 100;

NAMES-BUFFER (OUTPUT)
  {09}TABLE1.C1
```

Example 2:

```
STATEMENT
  SELECT C1 FROM TABLE1 T1 WHERE T1.C1 > 100;

NAMES-BUFFER (OUTPUT)
  {05}T1.C1
```


Example 3:

```
STATEMENT
  SELECT C1 FROM =TABLE_NAME_1 WHERE C1 > 100;

  NAMES-BUFFER (OUTPUT)
    {15}TABLE_NAME_1.C1
```

Example 4:

```
STATEMENT
  SELECT C1 FROM =TABLE_NAME_1 T1 WHERE T1.C1 > 100;

  NAMES-BUFFER (OUTPUT)
    {05}T1.C1
```

For information about the contents and use of a names buffer, see the SQL/MP programming manual for your host language.

`COLLATIONS INTO :collations-buffer`

specifies the name of a collations buffer declared in an INCLUDE SQLDA statement or elsewhere in your program into which DESCRIBE returns collations associated with the SELECT columns.

If you specify the COLLATIONS INTO clause, DESCRIBE sets the CPRL_PTR item for each entry in the SQLDA to the address of the corresponding entry in the collations buffer.

Each entry in the collation buffer is in VARCHAR format: the entry begins with a 2-byte numeric prefix that contains the length of the collation; the collation follows.

To compare collations associated with different objects, use the Guardian procedures described in the SQL/MP programming manual for your host language.

Example—DESCRIBE

This COBOL statement returns descriptions of output variables in the prepared statement identified by S1 to the SQLDA identified by :OUT-SQLDA:

```
EXEC SQL DESCRIBE S1 INTO :OUT-SQLDA
      NAMES INTO :OUT-NAMESBUF
      COLLATIONS INTO :OUT-COLLBUF END-EXEC.
```

The NAMES INTO and COLLATIONS INTO clauses direct DESCRIBE to return the names of the output variables and any collations associated with the output variables to buffers reserved for them. DESCRIBE sets the VAR-PTR and CPRL-PTR fields of each entry in the SQLDA to the addresses of the corresponding name and collation.

For examples of related statements and a detailed discussion of dynamic SQL programming techniques, see the *SQL/MP Programming Manual for C* or the *SQL/MP Programming Manual for COBOL*.

Detail Alias

A detail alias is a name assigned to a print item using the NAME clause of the DETAIL command. You can use a detail alias to refer to the print item in report formatting commands such as TOTAL and SUBTOTAL, but not within the DETAIL command itself.

A detail alias is not the same as an alias, which is a name assigned to a column in the select list of the SELECT command using the report writer NAME command. You can use an alias (but not a detail alias) to refer to a column in any part of the report definition.

DETAIL Command

DETAIL is an SQLCI report writer command that defines the contents and format of a detail line. (A detail line is the output from a single row returned by a SELECT command, although it can be printed as more than one row in a report.) You can use DETAIL only from the select-in-progress prompt, not from the SQLCI prompt.

```

DETAIL [ print-list ] ;

print-list is:

    print-item [ , print-item ] ... ]

print-item is:

    { { { column-id } [AS format] } [head] [name] }
      { literal }
      { num-exp }
      { CONCAT (print-list)[AS format] }
      { IF cond-expr THEN (print-list) }
      { [ ELSE (print-list) ] }
    }

    NEED [number]

    PAGE [number]

    SKIP [number]

    SPACE [number]

    TAB [number]

head is:

    HEADING "characters" [ CENTER ] | NOHEAD

name is:

    NAME detail-alias

```

column-id

is a column in the select list of the SELECT command to print in the detail line. *column-id* can be a column name, an alias, or COL *number* (which specifies the position of the column in the select list). It cannot be a detail alias.

literal

is a string literal to print in the detail line.

num-exp

is an numeric expression to evaluate and print in the detail line. The expression cannot include the AVG, COUNT, MIX, MAX, or SUM functions. It can specify columns with column names, aliases, or COL *number*, but not with detail aliases. It can include the report functions LINE_NUMBER, COMPUTE_TIMESTAMP, CURRENT_TIMESTAMP, and PAGE_NUMBER (which are not allowed in numeric expressions other than in print items).

For information about the form of numeric expressions, see [Expressions](#) on page E-21.

AS format

specifies a format for printing the item using the syntax of the AS clause or the AS DATE/TIME clause. For information, see [AS Clause](#) on page A-60 or [AUDIT File Attribute](#) on page A-74.

If you use items of date-time types with AS DATE/TIME, first convert them to Julian timestamps with JULIANTIMESTAMP. You cannot use AS with items of type INTERVAL.

CONCAT (*print-list*) [*AS format*]

concatenates print items. For description of this clause, see [CONCAT Clause](#) on page C-63.

IF *cond-expr* THEN (*print-list*)
[ELSE (*print-list*)]

specifies a conditional expression that determines whether to print the specified print list. For description of this clause, see [IF/THEN/ELSE Clause](#) on page I-1.

HEADING "*characters*" [CENTER] | NOHEAD

characters to use as a heading for the print item. The heading can contain any single or multibyte character.

To specify a multiline heading (up to 50 lines), include new-line characters in *characters* to indicate the beginning of each new line. (The default new-line character is slash (/). To change the default, see [NEWLINE_CHAR Option](#) on page N-4.)

By default, the heading is left justified. If you include CENTER, the heading is centered over the print item.

NOHEAD specifies that the print item has no heading.

If you specify neither HEADING nor NOHEAD, report writer uses the default heading in effect for the column. For an explanation of heading defaults, see Determining Headings later in this entry.

NAME *detail-alias*

specifies an SQL identifier unique among column names in the select list as the detail alias for the print item.

You can refer to a detail alias in other report formatting commands such as TOTAL and SUBTOTAL, but you cannot refer to a detail alias in the DETAIL command itself.

NEED [*number*]

advances to the next page before printing the next print item unless at least *number* (an integer in the range 1 through 32,767) more lines fit on the page. The default is 1. (When you compute the number of lines you need, include lines for the page footing.)

PAGE [*number*]

advances to the next page before printing the next print item, printing a page footing and page title, if defined.

If you specify *number* (an integer in the range 1 through 32,767), report writer uses *number* as the page number for the next page.

SKIP [*number*]

advances *number* (an integer in the range 1 through 32,767) lines before printing the next print item. For example, SKIP 1 causes the print item to be printed on the next line; SKIP 2 prints the item with one blank line. If fewer than *number* lines remain on the page, the report writer advances to the top of the next page.

A line is defined in terms of the spacing specified by the current LINE_SPACING option (see [LINE_SPACING Option](#) on page L-16). To determine the position of the next line, the report writer multiplies the value of LINE_SPACING by the value of SKIP.

Omitting *number* is equivalent to specifying SKIP 1.

SPACE [*number*]

prints *number* (an integer in the range 0 through 255) spaces between the specified print items. A space occupies one print position on the output line (one byte, regardless of the character set in use).

If you omit the entire clause, the default is the value of the SPACE option of the SET LAYOUT command. If you specify SPACE but omit *number*, the default is 1.

TAB [*number*]

prints the next print item at print position *number* of the output line. A print position occupies one byte, regardless of the character set in use.

number is an integer that corresponds to a print position between the left and right margins. The default is 1.

You can tab backward (specify a print position earlier in the line than the current print position), but if you overlap print items, the most recent item you specify overwrites older ones (or portions of older ones).

Considerations—DETAIL

- The detail line for the default report is:

```
DETAIL COL 1, COL 2, COL 3, ... ;
```

A DETAIL command with no print list (DETAIL;) produces a report that has no detail lines. This can be useful for summary reports.

- Only one DETAIL command is in effect at a time. Use the FC command to modify the current DETAIL command, or use the RESET REPORT command to reset the DETAIL line to the default.
- The output of the print list you specify in a DETAIL command is a logical line, although (depending on margin settings, device widths, and use of the SKIP clause) it might print on more than one physical line. A logical line is limited to 4072 bytes, including the field widths of all print items and the number of spaces between items.
- To create a report that includes only information about groups of records, use the BREAK ON command to specify groups, use the BREAK TITLE and BREAK FOOTING commands to specify the contents of the report lines, and use a DETAIL command with no arguments (DETAIL;) to suppress printing of detail lines. Only the summary lines will print.
- The report writer goes through these steps to determine the heading to use for a print item:
 1. Check the HEADINGS style option. If OFF, do not generate headings; if ON, continue.
 2. Check whether there is a DETAIL command for the current report that includes NOHEAD or HEADING clauses. If so, generate headings accordingly; if not, continue.
 3. Check for the most recent alias or detail alias. If an alias exists, use the alias as the heading; if not, continue.
 4. Check whether the table or view column has a heading defined for it. If so, use the heading; if not, use the column name as the heading.
 5. Generate a default (EXPR) heading for each expression, function, or numeric literal unless a heading was specified for the item in a DETAIL command.
 6. Omit headings for string literals and IF/THEN/ELSE or CONCAT items unless headings were specified for them in the DETAIL command.

Examples—DETAIL

- This example specifies a report in the format shown:

```
>> SET LIST_COUNT 0;
>> SELECT * FROM SALES.CUSTOMER ORDER BY CUSTNUM;
S> DETAIL CUSTNUM HEADING "Account",
+>   CUSTNAME AS A20 HEADING "Name" CENTER,
+>   CONCAT (CITY STRIP, ", ", STATE)
+>   HEADING "Location" CENTER,
+>   TAB 60, IF CREDIT = "A1" THEN ("***" );
```

Some sample lines of this report are:

Account	Name	Location
21	CENTRAL UNIVERSITY	PHILADELPHIA, PENNSYLVANIA ***
123	BROWN MEDICAL CO	SAN FRANCISCO, CALIFORNIA

- This example specifies a report in the format shown:

```
>> SELECT FIRST_NAME, LAST_NAME, DEPTNAME, MANAGER, AVG (SALARY)
+> FROM PERSNL.EMPLOYEE E, PERSNL.DEPT D
+> WHERE D.DEPTNUM = E.DEPTNUM
+> GROUP BY FIRST_NAME, LAST_NAME, DEPTNAME, MANAGER;
S> DETAIL DEPTNAME AS A20 HEADING "Dept." NAME DN,
+> CONCAT (FIRST_NAME STRIP, SPACE 1, LAST_NAME) AS A18
+> HEADING "Manager", COL 5 HEADING "Avg. Salary" NAME AVSAL;
```

Some sample lines of this report are:

Dept.	Manager	Avg. Salary
ASIA SALES	SHERRIE WONG	45835.00
ENGINEERING	ERIC BROWN	35000.00
MARKETING	JACK RAYMOND	30166.67

- This example demonstrates usage of SKIP to print a report vertically:

```
>> SET LIST_COUNT 0;
>> SET STYLE HEADINGS OFF,
>> SET LAYOUT PAGE_LENGTH ALL;
>> SELECT CUSTNAME, STREET, CITY, STATE, POSTCODE
+> FROM SALES.CUSTOMER
+> ORDER BY STATE, POSTCODE;
S> DETAIL PAGE,
+>   CUSTNAME, SKIP 1,
+>   STREET, SKIP 1,
+>   CONCAT (CITY STRIP, ", ", STATE STRIP,
+>   " ", POSTCODE);
+> LIST NEXT 1;
```

Some sample lines of this report are:

```
FRESNO STATE BANK
2300 BROWN BLVD
FRESNO, CALIFORNIA 93921
```

DISPLAY STATISTICS Command

DISPLAY STATISTICS is an SQLCI command that displays statistics about the last DML or PREPARE the statement you executed.

```
DISPLAY STATISTICS ;
```

Considerations—DISPLAY STATISTICS

- You can use DISPLAY STATISTICS at any time to see the statistics of the most recent DML or PREPARE statement but if you set the STATISTICS option to ON, SQLCI displays statistics automatically after each DML, DCL, DDL, or PREPARE statement executes. For information, see [SET SESSION Command](#) on page S-39.
- SQLCI displays these statistics:

- Estimated cost

Estimated cost is a relative measure of cost derived using the same cost functions that the SQL compiler uses to choose a query execution plan. The estimate includes processor, disk I/O, and message costs. Higher numbers tend to indicate that SQL expects to process larger amounts of data. Large numbers are not typically associated with fast on-line response times. Lower numbers imply more efficient execution, but can be affected by missing or inaccurate statistics or by skewed data distribution.

For information about cost, see the *SQL/MP Query Guide*.

- Start Time, End Time, Elapsed Time, and Master Executor Execution Time

Master Executor Execution time is the amount of processor time used by the SQL executor. Elapsed time includes the execution time, I/O time, and the time used to display the result.

- Number of records accessed and number of records used

Records accessed is the number of rows read (including rows that do not satisfy the selection criteria). Rows are counted for each table, underlying table of a protection view, and temporary table. If you join a table to itself, separate statistics are reported for each instance of the table. The number of rows accessed in an index is not reported.

Records accessed does not indicate the specific number of physical disk reads or writes because SQL uses disk caching to reduce the number of physical read and write operations.

Records used is the number of rows that satisfy the query.

- Number of disk reads
- Message count

Message count is usually the number of blocks passed from the disk process to the file system. Sometimes an additional message is needed to ensure that the last row was processed.

- Message bytes

Message bytes is the total amount of data transferred.

- Lock

Lock contains information about lock escalation.

Example—DISPLAY STATISTICS

The last DML command is:

```
>> DELETE FROM I.P
+> WHERE SUPPNUM NOT IN (SELECT SUPPNUM FROM I.S
+> WHERE STATE = "TEXAS");
--- 41 row(s) deleted.
```

Although you did not display the statistics when executed, you can display them using the DISPLAY STATISTICS command:

```
>> DISPLAY STATISTICS;
```

```
Estimated Cost          39
```

```
Start Time          94/06/01 09:07:16.678185
End Time            94/06/01 09:07:56.533061
Elapsed Time                00:00:39.854876
Master Executor Execution Time 00:00:00.359826
```

Table Name	Records Accessed	Records Used	Disk Reads	Message Count	Message Bytes	Lock
\S.\$V.I.S	48	42	4	84	6684	
\S.\$V.#2549	0	0	0	0	0	
\S.\$V.I.P	49	49	2	3	542	
\S.\$V.#2549	0	0	0	0		0

DISPLAY USE OF Command

DISPLAY USE OF is an SQLCI utility command that displays a list of SQL objects and registered SQL object programs depend on the object specified.

```
DISPLAY USE OF object [ AT { \node
                        [ { (\node [, \node] ... ) } ]
                        [ , { BRIEF | STANDARD } ] ] ;
```

object

is the name (or an equivalent DEFINE) of a collation, index, table, or view for which to specify the dependent objects.

If *object* is a secondary partition, you must include either its node or the node that contains the primary partition in the nodes you specify in the AT clause.

If SMF is installed on your node, *object* cannot specify any object on a \$*.ZYS*. subvolume.

```
AT { \node
    { ( \node [, \node ] ... ) }
```

specifies nodes to search for dependent objects and programs. The default is all nodes in the network.

```
{ BRIEF | STANDARD }
```

specifies the information and the format for display.

BRIEF Display name, object type, and status of dependent objects and programs.

STANDARD Display name, object type, status, owner, security, number of partitions, and catalog name for the object and for dependent objects and programs.

The default is STANDARD.

Considerations—DISPLAY USE OF

- DISPLAY USE OF requires authority to read the catalogs that describe the object you specify and the catalogs that describe all its dependent objects and programs.
- DISPLAY USE OF searches the USAGES table in the catalog that describes *object* to find the names of objects or programs that depend on *object*, then searches the catalogs that describe the dependent objects and finds the names of objects that in turn depend on the dependent objects. DISPLAY USE OF repeats this process until it finds all objects that depend directly or indirectly on *object*.

DISPLAY USE OF lists only registered SQL programs. Unregistered SQL programs cannot be listed because neither such programs nor their dependency relationships are described in catalogs.

The types of objects that can depend on each type of object are:

Objects	Dependent Objects
Table	Indexes, protection views, shorthand views, SQL programs
Protection view	Shorthand views, SQL programs
Shorthand view	Shorthand views, SQL programs
Index	SQL object programs
Collation	Indexes, protection views, shorthand views, SQL programs, tables

- The USAGES catalog table contains descriptions of the relations among primary partitions only. The only exception to this rule is that the USAGES table describes a relation between each partition of a protection view and the corresponding partition of the table that the protection view depends on. If an object depends on a protection view, however, the USAGES table describes only the relation between the primary partitions of the dependent object and the protection view.

If *object* is the name of a secondary partition, DISPLAY USE OF substitutes the name of the primary partition and reports the primary partitions of all the objects that depend on that primary partition.

- The columns of the DISPLAY USE OF output contain these information:

- Object Name

The name of the object or one of its dependent objects or programs, preceded by an integer level number. (The original object is at level 0, objects directly dependent on the original object are at level 1, and so forth.)

All names of dependent objects appear as fully qualified Guardian names. For an SQL program in an OSS file, the fully qualified Guardian name is the ZYQ name of the program file, and a pathname for the file appears immediately below the ZYQ name, wrapping over multiple lines if necessary. (If no pathname for the file is accessible to the user, the message “No pathname accessible.” appears instead.)

- Type

A two-character code that specifies the object type:

CP	Collation
IN	Index
PG	SQL program

PV	Protection view
SV	Shorthand view
TA	Table

- S

A status column that indicates whether any special condition was encountered during the search for the initial or dependent object. A blank indicates no special condition was encountered. The meanings of the status codes follow:

*	The object was listed previously; its dependent objects will not be repeated.
?	A system error occurred.
@	The AT option did not include this node.
I	The catalog for this object is marked “inconsistent.”
L	The object was not found.
N	The node with this object is not available.
R	The primary partition of this object was not found.
T	The type is unsupported.
U	The node is undefined.

- P

The number of partitions. A blank indicates that the object is not partitioned.

- The user ID of the object's owner.

- Secure

The security string for the object.

- Catalog Name

The name of the catalog in which the object is described.

The “Number of unique dependencies” field shows the number of objects that depend directly or indirectly on the initial object. The number does not include duplicate occurrences of objects.

The “Number of direct dependencies” field shows the number of objects that depend directly on the object being traced.

Examples—DISPLAY USE OF

- This example displays a BRIEF-format list of all objects and registered programs in the network that depend on the EMPLOYEE table in the subvolume \SYS1.\$VOL1.PERSNL. (The display has been modified slightly to fit the page width.)

```
>> DISPLAY USE OF \SYS1.$VOL.PERSNL.EMPLOYEE, BRIEF;
Object Name                                     Type S
-----
0 \SYS1.$VOL1.PERSNL.EMPLOYEE                 TA
1 \SYS1.$VOL1.PERSNL.EMPLIST                   PV
1 \SYS1.$VOL1.PERSNL.XEMPDEPT                  IN
1 \SYS1.$VOL1.PERSNL.XEMPNAME                  IN
1 \SYS1.$DATA.ZYQ39483.Z000002H                PG
```

```
PATH/usr/empinfo/reports/app.exe
U = Undefined node      N = Node unavailable
@ = Node not in list    * = Previously displayed
T = Unsupported type    ? = System error
Number of unique dependencies : 4
Number of direct dependencies : 4
```

- This example displays a STANDARD-format list of all objects and registered programs in the network that depend on a table named EMP on subvolume \SYS1.\$VOL1.HR. (The display layout has been modified slightly to fit the page width.)

```
>> DISPLAY USE OF \SYS.$VOL.PERSNL.EMPLOYEE;
Object Name                                     Type  S  P  Owner Name  Secure
-----
          Catalog Name
          -----
0 \SYS1.$VOL1.HR.EMP                           TA                                GROUP.NAME  CUCU
  $VOL1.HR
1 \SYS1.$VOL1.HR.EMPL                          PV                                GROUP.NAME  CUCU
  $VOL1.HR
1 \SYS1.$VOL1.HR.XEMP                          IN                                GROUP.NAME  CUCU
  $VOL1.HR
1 \SYS1.$DATA.ZYQ39483.Z000002H PG
  PATH/usr/empinfo/reports/app.exe
    OWNER:  GRP.FRED                                SECURITY -rwxr-x--x
    GROUP:  NonStop
          $VOL1.HR
1 \SYS1.$VOL1.HR.GPROG                         PG                                GROUP.NAME  CUCU
U = Undefined node      N = Node unavailable
@ =Node not in list    * = Previously displayed
T = Unsupported type    ? = System error
Number of unique dependencies : 4
Number of direct dependencies : 4
DISTINCT Clause
```

DISTINCT is a clause that removes duplicate rows from a result table. It is used in many search conditions and in the SELECT statement.

For information, see [SELECT Statement](#) on page S-18 or the entry for a specific search condition (for example, AVG or SUM).

DML Statements

A DML statement is used to select, update, insert, or delete rows in one or more tables. This table summarizes DML statements:

CLOSE Statement	Closes a cursor
DECLARE CURSOR Statement	Defines a cursor
DELETE Statement	Deletes rows from a table or view
FETCH Statement	Retrieves a row from a cursor
INSERT Statement	Inserts a row into a table or view
OPEN Statement	Opens a cursor
SELECT Statement	Retrieves data from tables and views
UPDATE Statement	Updates values in columns of a table or view

DOWNGRADE CATALOG Command

DOWNGRADE CATALOG is an SQLCI utility command that converts catalogs to an older version so the catalogs can be accessed by an older version of the SQL/MP software.

```
DOWNGRADE CATALOG[S] [ catalogs ] TO version ;
```

catalogs

specifies the catalogs to downgrade. *catalog* can be a single catalog name, a type CATALOG DEFINE name, or a name that specifies multiple catalogs by including these wild-card characters:

- ? matches any single character
- * matches 0 to 8 characters

For example,

- MYCAT? matches MYCATA, MYCATB, and MYCAT5 (and possibly others) but not MYCATXX.
- \$DATA.* matches all catalogs on volume \$DATA.
- =APPCAT matches the catalog specified by the =APPCAT DEFINE
- (MYCAT?, =APPCAT) matches MYCAT1, MYCAT2, MYCATX, (and possibly others) and the catalog specified by =APPCAT
- \$*.* matches all catalogs on the current default node except the system catalog.

Catalogs specified by *catalogs* can be either local or remote but cannot include objects with a version newer than *version*, or any catalog specified programs

with a PCV newer than *version*. (Delete any such objects or programs before you execute DOWNGRADE CATALOG.)

In addition, no catalog specified can include objects with a version newer than the version of the SQL/MP software executing the DOWNGRADE CATALOG command.

Catalogs cannot be a system catalog. (Use DOWNGRADE SYSTEM CATALOG to convert a system catalog.)

If SMF is installed on your node, *catalogs* cannot specify any catalog or system catalog on a *\$*.ZYS**. subvolume.

The default is the current default catalog.

TO *version*

specifies the catalog format version for the downgraded catalog.

You can express *version* as either an integer (2, 300, 310, 315, 320, 325, or 330) or a character string (A011, A300, A310, A315, A320, A325, or A330), but the version you specify must be older than the version of each catalog you specify with *catalogs*.

You cannot downgrade a catalog to version 1, but version 2 catalogs are compatible with version 1 software.

Considerations—DOWNGRADE CATALOG

- To downgrade a catalog, you must be a generalized owner of the catalog and you must have authority to read, write, and purge each table in the catalog. You also must have authority to write to the CATALOGS table in the system catalog.

DOWNGRADE CATALOG requires exclusive access to the catalogs being downgraded. Other processes cannot access the catalogs during the downgrade. The downgrade fails if another process has one of the catalogs open when you execute DOWNGRADE CATALOG.

If you downgrade a catalog to version 2, file labels must be available during the downgrade for any tables or objects registered in the catalog that have a nonzero value for the OBJECTVERSION column of the TABLES or INDEXES catalog table. (For backward compatibility, DOWNGRADE CATALOG changes such file labels to specify object version 0.)

- DOWNGRADE CATALOG invalidates any program that refers to a catalog table in the downgraded catalogs, but does not invalidate a program merely because the program is registered in a downgraded catalog or because it accesses an object (such as a user table) described in a downgraded catalog.

DOWNGRADE CATALOG does not invalidate a program registered in a downgraded catalog merely because the program has a PFV newer than *version* because such a program can execute regardless of the catalog downgrade unless

the SQL/MP software that executes it is replaced with an older version. However, if the purpose of the DOWNGRADE CATALOG is to prepare for installation of an older version of SQL/MP software, you will need to re-SQL-compile programs with a newer PFV after the older software is in place. (A runtime error occurs if you attempt to execute a program with a PFV newer than the version of the installed SQL/MP software.)

- DOWNGRADE CATALOG creates a new temporary catalog on the same volume as each catalog being downgraded. Such volumes must have enough disk space available to store files twice as large as the original catalog.
- You cannot use DOWNGRADE CATALOG in a user-defined transaction.
- An error that causes the downgrade of one catalog specified in *catalogs* to fail does not necessarily cause the downgrades of other catalogs specified in *catalogs* to fail. (Use GET VERSION to check the version of a specific catalog.)

In unusual failure situations (such as a system failure during a downgrade catalog operation), temporary files with names that begin with the letters “ZZDN” might be left on the same subvolume as the catalog. You can delete these with CLEANUP.

Examples—DOWNGRADE CATALOG

- This example downgrades the catalog on the subvolume \$VOL1.SVOL1 to version 2:

```
>> DOWNGRADE CATALOG $VOL1.SVOL1 TO 2;
```
- This example downgrades all the catalogs on volume \$VOL to version 310:

```
>> DOWNGRADE CATALOG $VOL.* TO 310;
```
- This example downgrades all the catalogs on the current default node to version 2:

```
>> DOWNGRADE CATALOG $*.* TO 2;
```
- This example downgrades all catalogs on a volume on a remote node to version 2:

```
>> DOWNGRADE CATALOG \DIST.$DATA.* TO A011;
```


DOWNGRADE SYSTEM CATALOG Command

DOWNGRADE SYSTEM CATALOG is an SQLCI utility command that allows a user with super ID authority to convert a local system catalog to an older version so the system catalog can be accessed by an older version of SQL/MP software.

```
DOWNGRADE SYSTEM CATALOG TO version ;
```

version

is the catalog format version for the downgraded system catalog.

You can express *version* as either an integer (2, 300, 310, 315, 320, 325, or 330) or a character string (A011, A300, A310, A315, A320, A325, or A330), but the version you specify must be older than the current version of the system catalog.

You cannot downgrade a catalog to version 1, but version 2 catalogs are compatible with version 1 software.

Considerations—DOWNGRADE SYSTEM CATALOG

- Only the local super ID can downgrade a system catalog.

DOWNGRADE SYSTEM CATALOG requires exclusive access to the system catalog. Other processes cannot access the system catalog during the downgrade. The downgrade fails if another process has one of the catalog tables in the system catalog open when you execute DOWNGRADE SYSTEM CATALOG.

The downgrade fails, If you downgrade a system catalog to version 2, file labels must be available during the downgrade for any tables or objects registered in the catalog that have a nonzero value for the OBJECTVERSION column of the TABLES or INDEXES catalog table. (For backward compatibility, DOWNGRADE CATALOG changes such file labels to specify object version 0.)

- You cannot downgrade a system catalog that contains protection views, objects with a version newer than *version*, or programs with a PCV newer than *version*. (Drop any such objects or programs before you execute DOWNGRADE SYSTEM CATALOG.) In addition, a system catalog cannot itself have a version newer than the version of the SQL/MP software executing the DOWNGRADE SYSTEM CATALOG command.

You can downgrade a system catalog that has higher-version user catalogs registered in it. For example, you can downgrade a version 310 system catalog to version 2, even if it has version 310 user catalogs registered in it.

- DOWNGRADE SYSTEM CATALOG invalidates any program that refers to a catalog table in the downgraded system catalog, but does not invalidate a program merely because the program is registered in a downgraded system catalog or accesses an object (such as a user table) described in a downgraded system catalog.

DOWNGRADE SYSTEM CATALOG does not invalidate a program registered in the downgraded system catalog that has a PFV newer than *version* because such a program can execute regardless of the catalog downgrade unless the SQL/MP software that executes the program is replaced with an older version. However, if the purpose of the DOWNGRADE SYSTEM CATALOG is to prepare for installation of an older version of SQL/MP software, you will need to re-SQL-compile programs with a newer PFV after the older software is in place. (A runtime error occurs if you attempt to execute a program with a PFV newer than the installed SQL/MP software.)

- DOWNGRADE SYSTEM CATALOG creates a new temporary catalog on the same volume as the catalog being downgraded. Such volumes must have enough disk space available to store files twice as large as the original system catalog.
- You cannot use DOWNGRADE SYSTEM CATALOG in a user-defined transaction.
- In unusual failure situations (such as a system failure during the downgrade operation), temporary files with names that begin with the letters “ZZDN” might be left on the same subvolume as the catalog. You can delete these with CLEANUP.

Example—DOWNGRADE SYSTEM CATALOG

This command downgrades the local system catalog to version 2:

```
>> DOWNGRADE SYSTEM CATALOG TO 2;
```

DROP Statement

DROP is a DDL statement that deletes a catalog, collation, constraint, index, SQL-program Guardian file, table, or view, and deletes comments associated with the dropped object.

```
DROP { CATALOG [ catalog ]
      { COLLATION collation
      { CONSTRAINT constraint ON table
      { INDEX index
      { PROGRAM file
      { TABLE table
      { VIEW view
      }
```

CATALOG [*catalog*]

specifies the name (or an equivalent DEFINE) of an empty catalog to delete. If you omit the catalog name, SQL deletes the default catalog.

COLLATION *collation*

specifies the name (or an equivalent DEFINE) of a collation to delete.

CONSTRAINT *constraint* ON *table*

specifies the name of a constraint to delete, and the name (or an equivalent DEFINE) of the table with which the constraint is associated.

INDEX *index*

specifies the name (or an equivalent DEFINE) of an index to delete. *index* cannot be a catalog index.

PROGRAM *file*

specifies the name (or an equivalent DEFINE) of a Guardian file that contains an SQL program.

TABLE *table*

specifies the name (or an equivalent DEFINE) of a table to delete. *table* cannot be a catalog table.

VIEW *view*

specifies the name (or an equivalent DEFINE) of a view to delete.

Considerations—DROP

- DROP requires authority to read and write to the catalog that describes the object, and to read and write to the catalogs of related objects that require changes

because of the drop. In addition, you cannot drop an object until after the time and date specified for the NOPURGEUNTIL attribute of the object.

To drop a table, view, or program, you must also have authority to purge the object being dropped. To drop a partitioned object, all partitions must be accessible. Additional requirements for dropping other types of objects are described later in this entry.

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 execute a DROP statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

△ **Caution.** Using the DROP statement to delete a table is permitted if you have PURGE authority, there are no locks outstanding on the table, and if you are beyond the NOPURGEUNTIL date.

To adequately protect important tables, use the NOPURGEUNTIL attribute, specifying a date well into the future, or change PURGE authority to “-” (SUPER only). Either method requires an ALTER TABLE before the DROP TABLE.

- You cannot drop a catalog unless you first drop all user tables, views (except for views defined on catalog tables), indexes, collations, and program files from the catalog. To drop a catalog, you must have authority to read and purge the catalog tables, and authority to read and write to SQL.CATALOGS.
- You cannot drop a collation that has dependent objects. To learn how to determine dependencies, see [DISPLAY USE OF Command](#) on page D-54.
- Dropping a constraint invalidates all SQL object program files that use the underlying table and can change the version of the associated table and any views defined on that table.

To drop a constraint, you must be the local or remote owner of the underlying table, with purge authority, or the local super ID. You cannot drop a constraint unless the underlying table (including all partitions) and the catalogs of all SQL object program files that use the underlying table are accessible.

- Dropping an index purges the physical file for the index, including all its partitions. It also invalidates all SQL object program files that use the table underlying the index and it can change the version of the table and any views defined on the table.

To drop an index, you must be the local or remote owner of the underlying table, with purge authority, or the local super ID. You cannot drop an index unless the

table (including all partitions) and the catalogs of all SQL object program files that use the table are accessible.

-
- △ **Caution.** Using the DROP statement to delete a table is permitted if you have PURGE authority, there are no locks outstanding on the table, and if you are beyond the NOPURGEUNTIL date.

To adequately protect important tables, use the NOPURGEUNTIL attribute, specifying a date well into the future, or change PURGE authority to “-” (SUPER only). Either method requires an ALTER TABLE before the DROP TABLE.

- You cannot drop a table or view unless all related indexes and views (including all partitions) are accessible, and unless the catalogs of all SQL object program files that use the table or view are accessible. You cannot drop a protection view unless the underlying table (including all partitions and associated indexes) is accessible. To make sure the object is accessible, perform a LOCK TABLE operation before issuing the DROP TABLE request.

Dropping a table purges the physical file for the table and drops all dependent indexes, constraints, partitions, and views. It also invalidates all SQL object program files that refer to the table. If the dropping process does not have authority to purge a shorthand view, SQL invalidates the shorthand view.

Dropping a view automatically drops all dependent views, and invalidates all SQL object program files that refer to the views. If the dropping process does not have authority to purge a shorthand view, SQL invalidates the shorthand view.

Invalidated shorthand views are unusable and should be dropped by a user with appropriate security.

Invalidated program files might be usable, but you should ensure that they are explicitly SQL-compiled to avoid automatic recompilation each time the program runs.

-
- △ **Caution.** Using the DROP statement to delete a table is permitted if you have PURGE authority, there are no locks outstanding on the table, and if you are beyond the NOPURGEUNTIL date.

To adequately protect important tables, use the NOPURGEUNTIL attribute, specifying a date well into the future, or change PURGE authority to “-” (SUPER only). Either method requires an ALTER TABLE before the DROP TABLE.

- You cannot drop a nonaudited object within a user-defined TMF transaction.
- Dropping a program purges the physical program file.

To drop a program, you must have authority to read and write to the USAGES table of each catalog that contains objects referenced by the program.

You cannot use DROP to delete an SQL-program OSS file. Use the OSS rm command or the OSS unlink() function instead. (These commands remove the use of one pathname for a file. OSS removes the physical SQL object program file when the last pathname is removed; SQL deletes any references to the program in the SQL catalog at that time.)

You cannot drop an SQL-program Guardian file within a user-defined transaction. (You can use the OSS unlink command to delete an SQL-program OSS file within a user-defined transaction, but the operation is not performed as part of the transaction.)

- If SMF is installed on your node, the object to drop must have either a virtual or direct name.

This restriction does not apply to DROP CONSTRAINT.

Example—DROP

This example drops an index on the PARTS table and then drops the table and a related program:

```
DROP INDEX $VOL1.SALES.XPARTDES;
DROP TABLE $VOL1.SALES.PARTS;
DROP PROGRAM $VOL3.SUBVOL3.PROGA;
```

DROP SYSTEM CATALOG Command

DROP SYSTEM CATALOG is an SQLCI command that allows the local super ID to delete the system catalog, including the CATALOGS table and information about the SQLCI2 program.

```
DROP SYSTEM CATALOG catalog-name ;
```

catalog-name

identifies the system catalog to be dropped; *catalog-name* is the name of the volume and subvolume that contain the system catalog (or an equivalent DEFINE).

If SMF is installed on your node, the system catalog must have either a virtual or direct name.

Considerations—DROP SYSTEM CATALOG

- To drop the system catalog, you must use the local super ID.
- The system catalog must be empty except for the system catalog entry in the CATALOGS table and entries for the catalog tables and the SQLCI2 program. The system catalog must be the only catalog registered in the CATALOGS table. You must drop all other objects, programs, and catalogs before executing this command.
- If you are running SQLCI and have not entered any SQLCI commands during the current session, you can drop the system catalog by entering this command at the SQLCI prompt:

```
>> DROP SYSTEM CATALOG catalog-name ;
```

You cannot enter the DROP SYSTEM CATALOG command, however, while the SQLCI2 program is running, as it normally is after you enter a command during the current SQLCI session. If you attempt to enter the DROP SYSTEM CATALOG command while SQLCI2 is running, the command terminates abnormally and an error message appears.

To enter the command correctly, you can use either of two methods:

- Exit from SQLCI. Then restart SQLCI and enter the DROP SYSTEM CATALOG command at the first SQLCI prompt:

```
30> SQLCI
>> DROP SYSTEM CATALOG catalog-name;
```

- Enter the DROP SYSTEM CATALOG command at the TACL prompt:

```
31> SQLCI DROP SYSTEM CATALOG catalog-name;
```

Examples—DROP SYSTEM CATALOG

- This example drops a system catalog that resides on the default location \$SYSTEM.SQL:

```
16> SQLCI DROP SYSTEM CATALOG $SYSTEM.SQL;
```

- This example drops a system catalog that resides on the volume \$VOL1 and subvolume SVOL1:

```
18> SQLCI DROP SYSTEM CATALOG $VOL1.SVOL1;
```

DSL Statements

A DSL statement retrieves status information about the version of the database. This table summarizes DSL statements:

<u>GET CATALOG OF SYSTEM Statement</u>	Retrieves the name of a local or remote system catalog
<u>GET VERSION Statement</u>	Retrieves the version of a specific SQL object, catalog, or system
<u>GET VERSION OF PROGRAM Statement</u>	Retrieves the PCV, PFV, or HOSV of an SQL program

For detail, refer to the specific statement.

DSLACK File Attribute

DSLACK is a Guardian file attribute that specifies the minimum percentage of space to leave for future insertions when loading data blocks. DSLACK applies only to key-sequenced tables and to indexes.

DSLACK *percent*

The default is the value of the SLACK file attribute.
The default for SLACK is 15 percent.

percent

is an integer from 0 to 99 that specifies the percent of empty space to leave in each data block when loading the file.

Considerations—DSLACK

- DSLACK specifications are usually between 15 and 25 percent.
- Specifying a larger-than-normal DSLACK value when a file is initially loaded and many more inserts are expected can improve performance by reducing the number of block splits required when inserts occur.
- For a file expected to have little write activity, you can save disk space by specifying a smaller-than-normal DSLACK value.

DUP Command

DUP is an SQLCI utility command that copies tables (optionally with the associated views and indexes), views, collations, SQL programs in Guardian files, and Enscribe files. DUP cannot copy a catalog table.

DUP is useful for moving tables to different nodes or volumes and for duplicating tables for testing.

DUP resembles the FUP DUP command in function and syntax, but you cannot use FUP DUP on SQL objects.

```
DUP source-fileset-list,
    {
        target-fileset
        MAP NAME[S] { map-spec
                     { (map-spec [, map-spec] ... ) } }
        [ [,] dup-option ] ... ;

dup-option is:
{ CATALOG[S] { catalog-spec
               { ( catalog-spec [, catalog-spec ]...) } }
  COLLATION[S] ( collation-spec[,collation-spec]... )
  ALLOWERRORS [ ON | OFF | num ]
  [ NO ] LISTALL
  SAVEALL [ ON | OFF ]
  SAVEID [ ON | OFF ]
  SOURCEDATE [ ON | OFF ]
  [ TARGET ] { NEW | KEEP | PURGE }
  INDEX[ES] [ IMPLICIT | OFF ]
  VIEW[S] [ IMPLICIT | EXPLICIT | OFF ]

map-spec is:
    simple-fileset-list TO fileset

catalog-spec is:
    catalog-name [ FOR simple-fileset-list ]

collation-spec is:
{ collation-name
  { ( collation-name [ , collation-name ] ... ) }

    FOR simple-fileset-list
```

source-fileset-list

is a qualified fileset list that specifies the objects or files to duplicate. For information, see [Filesets](#) on page F-29.

If *source-fileset-list* includes a primary partition, DUP duplicates all partitions of the table or file, deriving names for new secondary partitions from the values you specify for *target-fileset* or the MAP NAME option. If *source-file-list* explicitly specifies a secondary partition, DUP reports an

error. If *source-file-list* implicitly specifies secondary partitions, DUP ignores the secondary partitions.

If SMF is installed on your node, *source-fileset-list* must not specify an object or file on a *\$*.ZYS** subvolume.

Duplication of views and indexes depends on the INDEXES and VIEWS options described later in this entry.

target-fileset

is a fileset that specifies names and locations for the new objects and files.

An asterisk (*) in the fileset specification indicates that the portion of the name in which the asterisk appears should be the same as the corresponding portion of the name of the object or file being duplicated. (The meaning of the asterisk differs from the usual meaning of an asterisk in a fileset specification. The ? (question mark) normally allowed in a fileset specification is not allowed.)

```
DUP $VOL1.SALES.*, $VOL2.*.*
```

duplicates each table and its dependent objects on \$VOL1.SALES, creating the duplicates on \$VOL2.SALES with the same names as the original tables and dependent objects.

For information, see the *target-fileset* specification Considerations item on page [D-76](#).

```
MAP NAME[S] { map-spec
              { (map-spec [, map-spec] ... ) } }
```

is a clause that specifies names and locations for the new objects or files.

map-spec is:

```
simple-fileset-list TO fileset
```

simple-fileset-list

is a simple fileset list that specifies the objects or files being duplicated for which names and locations are being specified.

A fileset in the list that does not specify a node matches any node. Specifying the local node for a fileset is equivalent to not specifying any node for the fileset. For example, if \LOCAL is the local node name, both the node specifications **.*.** and *\LOCAL.*.*.** match all files on all nodes, both local and remote.

fileset

is a fileset that specifies names and locations for the new objects and files.

An asterisk (*) in the fileset specification indicates that the portion of the name in which the asterisk appears should be the same as the corresponding portion of the name of the object or file being duplicated. For example,

```
MAP NAMES $WHS2.INVENT.PARTLOC TO $TEST.*.*
```

specifies that the new table that duplicates \$WHS2.INVENT.PARTLOC is to be \$TEST.INVENT.PARTLOC.

(The meaning of the asterisk differs from the usual meaning of an asterisk in a fileset specification. The ? normally allowed in a fileset specification is not allowed.)

If you specify a list of *map-specs* and one conflicts with another, DUP uses the first in the list.

For information about using MAP NAME, see the *target-fileset* specification Considerations item on page [D-76](#).

```
CATALOG[S] { catalog-spec
             { ( catalog-spec> [, catalog-spec ]...) } }
```

specifies existing catalogs in which the target objects are to be described;
catalog-spec is

```
catalog-name [ FOR simple-fileset-list ]
```

catalog-name identifies a catalog on the same node as the objects to hold the descriptions of the objects.

FOR *simple-fileset-list* specifies the names and location of the target objects to be described in the catalog. A fileset within *simple-fileset-list* that does not specify a node matches any node. Specifying the local node for a fileset is equivalent to not specifying any node for the fileset. For example, if \LOCAL is the local node name, both the node specifications *.* and \LOCAL.*.* match all files on all nodes, both local and remote.

If you omit the FOR clause, SQL uses *catalog-name* as the catalog for all duplicated objects.

If you omit the CATALOGS option, SQL uses the current default catalog.

```
COLLATION[S] ( collation-spec [, collation-spec ] ... )
```

specifies collations to be used by new objects:

```

{ collation-name }
{
  ( collation-name [ , collation-name ] ... ) }
FOR simple-fileset-list

```

collation-name identifies a collation and FOR *simple-fileset-list* specifies a simple fileset list that includes the names of any new objects that reference the collation.

The first collation specified whose simple name matches the simple name of a collation referenced in the new object being created is mapped to the new object. If the new object is referenced in more than one specified fileset, the first fileset specified is used. If no collation names match, or if no specified fileset contains the new object, no mapping occurs.

If you omit the COLLATION option, a new object that uses a collation references the same collation referenced by the original object.

ALLOWERRORS [ON | OFF | *num*]

specifies error handling:

- ON attempts to duplicate all specified files, regardless how many errors are encountered
- OFF stops the DUP operation after the first error is encountered
- num* duplicates all specified objects and files until the number of errors encountered exceeds the value of *num*

If you omit the ALLOWERRORS clause completely, the default is ALLOWERRORS OFF. If you specify ALLOWERRORS but do not specify an option, the default is ALLOWERRORS ON.

[NO] LISTALL

specifies whether to display the name of each duplicated file or object in this format:

```

DUPLICATED object-type source-name TO target-name
          PARTS (part-num,src-$volume TO tgt-$volume,
          ...
)

```

object-type is TABLE, INDEX, COLLATION, PVIEW, SVIEW, or FILE.

Dependent objects that are duplicated automatically when the underlying object is duplicated are listed below the underlying object. The PARTS clause appears only for partitioned files, tables, and indexes.

If you omit the LISTALL option, LISTALL is the default. If you specify NO LISTALL, DUP suppresses the display of confirmations.

SAVEALL [ON | OFF]

specifies a setting for both the SAVEID and the SOURCEDATE options.

- ON Set SAVEID and SOURCEDATE to ON
- OFF Set SAVEID and SOURCEDATE to OFF

If you specify SAVEALL and specify either SAVEID or SOURCEDATE separately, the settings must match.

The default is SAVEALL OFF.

SAVEID [ON | OFF]

specifies the security and owner for new objects and files:

- ON Set security and owner of each new item to that of the corresponding original item
- OFF Set the security of each new item to the default security of the user who executes DUP; make that user the owner

The default is SAVEID OFF.

SOURCEDATE [ON | OFF]

controls the timestamps of targets:

- ON Assign each item the timestamps of the corresponding original item (for tables: last modified, last opened, and most recently redefined; for Enscribe files, only last modified timestamps)
- OFF Assign each item timestamps for the date and time of the DUP operation

The default is SOURCEDATE OFF.

[TARGET] { NEW | KEEP | PURGE }

specifies what to do if a new item created by the DUP operation has the same name as an existing object or file in the target location:

- NEW Do not copy the item; report an error
- KEEP Do not copy the item; do not report an error
- PURGE Purge the object or file that has the same name as the new item, then create the new item as specified in the DUP command

An error occurs if you specify TARGET PURGE and the one of the two items that would have duplicate names is an SQL object and the other is an Enscribe file. An error also occurs if you specify TARGET PURGE and the existing item is a collation that has dependent objects.

The default is TARGET NEW.

```
INDEX[ES] [ IMPLICIT | OFF ]
```

specifies whether to duplicate indexes of duplicated tables:

IMPLICIT	Duplicate indexes of duplicated tables
OFF	Do not duplicate indexes of duplicated tables

The default is INDEXES IMPLICIT.

```
VIEW[S] [ IMPLICIT | EXPLICIT | OFF ]
```

specifies whether to duplicate views of duplicated tables:

IMPLICIT	Duplicate a view only if one of its underlying tables is duplicated
EXPLICIT	Duplicate views only if the view names are in <i>source-fileset-list</i> , not merely because one of the underlying tables is duplicated.
OFF	Do not duplicate views.

The default is VIEWS IMPLICIT.

An error occurs if you specify VIEWS IMPLICIT, either explicitly or by default, and the *source-fileset-list* includes only some of the tables underlying a shorthand view.

Considerations—DUP

- DUP requires authority to read the objects and files being duplicated, authority to read catalogs in which the objects are described, authority to write to the catalogs in which new objects are to be described, authority to write to any files created, and authority to purge any objects or files that must be purged.
- DUP creates objects and files that have the same physical attributes as the original objects and files. In addition, DUP enforces these rules for various types of objects and files:
 - Rules for all SQL objects
 - If the catalog you specify to hold the description of a new object does not exist, DUP does not duplicate the object. Depending on the setting of ALLOWERRORS, this condition might cause the DUP operation to fail.
 - Any comments on a source object recorded in the COMMENTS table of the original catalog are applied to the new object and recorded in the COMMENTS table of the new catalog.
 - Rules for tables
 - *target-fileset* or the MAP NAME option you specify must include sufficient information to allow DUP to map names of any partitions, views,

or indexes being duplicated as part of duplicating a table. For an example, see the *target-fileset* specification Considerations item on page [D-76](#).

- If constraints or statistics exist for a table that is duplicated, DUP applies the constraints and statistics to the new table and records them in the appropriate catalog tables of the catalog for the new table.
- Rules for views
 - *target-fileset* or the MAP NAME option you specify determines how DUP translates table and view names (in the FROM clause of the SELECT command in the view definition) to the new view definition.
 - You can explicitly specify the name of a protection or shorthand view in *source-fileset-list* if you also specify VIEWS EXPLICIT.
 - The new copy of a protection view that you duplicate explicitly must reside on the same volume and be defined in the same catalog as the new table on which it is defined.
 - DUP does not duplicate shorthand views built on top of protection views unless you specify VIEWS EXPLICIT.
 - Because a shorthand view definition might reference tables and views that have not yet been duplicated, such a definition is invalid when DUP initially creates it. After duplicating all objects involved in the DUP operation, DUP attempts to validate the shorthand view definition. If the DUP operation does not complete for any reason, or if you specify an invalid mapping scheme, a view definition might be left in an invalid state.
- Rules for SQL program files
 - DUP does not register duplicated programs in an SQL catalog. Unless a duplicated SQL program was compiled with the NO REGISTER option, you must execute SQLCOMP to register the program in an SQL catalog before you can execute the program.
 - If the original program was compiled with the CHECK INOPERABLE plans option and referenced tables and views have the SIMILARITY CHECK option enabled, you can use the REGISTER ONLY option of the SQLCOMP command to register the program through SQLCOMP without recompiling it. If not, you must SQL-compile the program again. For information, see the SQL/MP programming manual for your host language.
 - DUP cannot duplicate an SQL program in an OSS file. An error occurs if you specify an OSS program in *source-fileset-list*.
- Rules for collations
 - When you specify both collations and other SQL objects in *source-fileset-list* and any of the other SQL objects references one or more of the collations, DUP copies the collations before the SQL

objects. If collations are the only objects in *source-fileset-list*, DUP copies the collations and then updates the CPRULES and CPRLSRCE catalog tables.

- Rules for Enscribe files
 - If an Enscribe file references its alternate-key files, DUP modifies such references in the new file based on the MAP NAME option you specify. If you omit the MAP NAME option, DUP does not modify the references.
 - To duplicate an alternate-key file, you must specify it explicitly or implicitly (through wild-card characters) in *source-fileset-list*.
 - DUP does not duplicate the DDL record definition of an Enscribe file.
- With some minor exceptions, a *target-fileset* specification is equivalent to this MAP NAME option:

```
MAP NAME *.*.* TO target-fileset
```

These are some of the guidelines for specifying *target-fileset* or MAP NAME options:

- If you want new objects to have the same unqualified object names as the original objects, specify *target-fileset* with an asterisk for the object part of the Guardian name, using specific names only for the parts of the name that are to be different, for example:

DUP \$VOL1.*.* , \$NWWVOL.*.*;	To new volume
DUP \$VOL1.SUBV1.* , *.NWSUBV.*;	To new subvolume
DUP \$VOL1.SUBV1.* , \$NWWVOL.NWSUBV.*;	To new volume and new subvolume
- If you want to give new unqualified names to the new objects or to map more than one name at the same level (node, volume, or subvolume) to a different name, you must use MAP NAME. For example, this command duplicates the PERSNL and SALES subvolumes to the same volume, but different subvolumes:

```
DUP ( $VOL1.PERSNL.* FROM CATALOG $VOL1.PERSNL ,
      $VOL1.SALES.* FROM CATALOG $VOL1.SALES ) ,
MAP NAME ( $VOL1.PERSNL.* TO $VOL1.NWPERS.* ,
           $VOL1.SALES.* TO $VOL1.NWSALES.* ) ;
```

- Do not map a table to a specific table name if you want its dependent objects (protection views and indexes) duplicated. This syntax is incorrect:

```
DUP PERSNL.EMPLOYEE , TESTSUBV.EMPLOYEE
```

This is the correct syntax:

```
DUP PERSNL.EMPLOYEE , TESTSUBV.*
```


- Be careful that you define the MAP NAMES and CATALOGS parameters correctly for dependent tables, indexes, views, and programs. An incorrect mapping scheme can leave the objects invalid or cause the RESTORE process to fail.

For example, suppose that base table \$A.A.TABLE has two dependent objects: a protection view located on \$A.XX.PVIEW and an index located on \$A.ZZ.IXTAB. To copy the base table and all dependent objects, you must use a MAP NAMES option that includes all the dependent objects. This MAP NAMES clause includes all three objects:

```
MAP NAMES ( $A.A.* TO $D.A.* ,
            $A.XX.* TO $D.XX.* ,
            $A.ZZ.* TO $D.ZZ.* )
```

When changing the catalog of the base table, you must also use a complete CATALOG mapping for all objects.

- Duplicating shorthand views might produce unexpected results. For example, this command produces a view that is identical to the source, but because the command does not specify naming patterns with wild-card characters, DUP cannot map the named objects in the view definition:

```
DUP MGRLIST, MGRLIST2, VIEW EXPLICIT;
```

This command does not work because DUP cannot map all the objects in the definition of the view MGRLIST to MGRLIST2:

```
DUP MGRLIST, MAP NAMES *.*.* TO MGRLIST2
```

This command duplicates the view as intended (assuming that MGRLIST references only EMP and DEPT, and that neither EMP nor DEPT is partitioned or has indexes):

```
DUP (MGRLIST, EMP, DEPT),
    MAP NAMES (MGRLIST TO MGRLIST2,
               EMP TO EMP2,
               DEPT TO DEPT2),
    VIEW EXPLICIT;
```

- To duplicate a table partitioned over multiple nodes, you must use the MAP NAMES and CATALOGS clauses, and you must specify the remote nodes first within each clause. (You must not specify the local node first because DUP ignores a local node specification in these clauses and the resulting fileset expression matches all nodes.)

For example, this command duplicates the PARTS table, which is partitioned over two nodes. \LOCAL is the node where the DUP command executes and \REMOTE is the remote node. The PARTS table partitions are duplicated to the same volumes and subvolumes as the original table, but with different names.

```
DUP $VOL1.TESTSUBV.PARTS,
MAP NAMES (
    \REMOTE.*.*.* TO \REMOTE.*.*.OLDPARTS,
```

```

    *.*.* TO *.*.OLDPARTS )
CATALOGS (
    \REMOTE.$VOL1.CAT FOR \REMOTE.*.*.*,
    $VOL1.CAT FOR *.*.* );

```

This command does not work as expected because DUP changes the \LOCAL.*.* part of the MAP NAMES specification to *.*.*. Then, because the local specification is given first, both local and remote partition names map to \LOCAL.*.*.OLDPARTS.

```

DUP $VOL1.TESTSUBV.PARTS,
MAP NAMES (
    \LOCAL.*.*.* TO \LOCAL.*.*.OLDPARTS,
    \REMOTE1.*.*.* TO \REMOTE1.*.*.OLDPARTS
)
CATALOGS (
    \LOCAL.$VOL1.CAT FOR \LOCAL.*.*.*,
    \REMOTE1.$VOL1.CAT FOR \REMOTE1.*.*.*
);

```

- You cannot execute the DUP command within a user-defined TMF transaction. Because it is not possible to duplicate data to an audited table, DUP creates nonaudited tables. If the original table is audited, DUP sets the new table's AUDIT attribute to ON after the DUP operation finishes.

If you want to be able to perform a TMF file recovery operation, make online dumps of new audited objects and files after the DUP. For information about performing dumps, see the *TMF Operations and Recovery Guide*.

You can press the Break key to interrupt a DUP operation. If you press the Break key, the operation stops. Items already duplicated remain, but the item being duplicated when you pressed Break can be left in an invalid state.

If DUP fails or if you press the Break key, the corrupt flag is set for the last object operated on before the failure. Run FILEINFO on all the target objects to find the corrupt objects, then delete the objects before restarting DUP.

If a DUP operation fails after correctly duplicating some objects and files, restarting the operation from the beginning causes errors unless the original operation used the TARGET PURGE option. Determine the objects and files that were duplicated and consider the appropriate setting for the TARGET option before you restart a partially successful DUP operation.

If an unusual situation occurs while a collation is being duplicated, two temporary files (ZZCL $nnnn$ and ZZCS $nnnn$, where $nnnn$ is a number) might be left in the same subvolume as the source collation. You can purge both of these files with the PURGE command.

Examples—DUP

- This example copies table \$VOL1.PERSNL.JOB to subvolume \$NEWVOL.PERSNL and gives it the same name as the original table. The new

table will be described in the current default catalog because the CATALOG option is not specified. The JOB table has no dependent objects.

```
>> DUP $VOL1.PERSNL.JOB, $NEWVOL.PERSNL.JOB LISTALL;
```

This command does the same thing:

```
>> DUP $VOL1.PERSNL.JOB, $NEWVOL.*.* LISTALL;
```

- This example copies table \$VOL1.PERSNL.JOB to the remote volume \SYS2.\$NEWVOL, using the same table name as the original. The CATALOG option specifies a catalog on the remote node in which to describe the new table.

```
>> DUP $VOL1.PERSNL.JOB, \SYS2.$NEWVOL.PERSNL.JOB,
+> CATALOG \SYS2.$NEWVOL.CAT FOR \SYS2.$NEWVOL.PERSNL.JOB;
```

- This example copies a partitioned table, PARTLOC, that has a primary partition on \$VOL1 and secondary partitions and indexes on \$WHS2 and \$WHS3. PARTLOC has an index, PARTIX, and a dependent protection view, PARTVW. The example copies PARTLOC to subvolumes of the same name on three different volumes, copying PARTIX and PARTVW also, because the default options INDEXES IMPLICIT and VIEWS IMPLICIT apply.

```
>> DUP $VOL1.INVENT.PARTLOC, *.TESTINV.*;
DUPLICATED TABLE $VOL1.INVENT.PARTLOC TO
$VOL1.TESTINV.PARTLOC
      PARTS (1, $WHS2 TO $WHS2,
            2, $WHS3 TO $WHS3)
      INDEX $VOL1.INVENT.PARTIX TO $VOL1.TESTINV.PARTIX
      PVIEW $VOL1.INVENT.PARTVW TO $VOL1.TESTINV.PARTVW
```

- This example copies the partitioned table from the previous example to a different node, \NEWSYS, by using the MAP NAMES option. Objects on \$VOL1 are placed on \$NVOL1, objects on \$WHS2 are placed on \$NVOL2, and objects on \$WHS3 are placed on \$NVOL3. Subvolume and table names remain the same. The CATALOG clause specifies a catalog on \NEWSYS in which to describe the new objects.

```
>> DUP $VOL1.INVENT.PARTLOC,
+> MAP NAME ( $VOL1.*.* TO \NEWSYS.$NVOL1.*.* ,
+> $WHS2.*.* TO \NEWSYS.$NVOL2.*.* ,
+> $WHS3.*.* TO \NEWSYS.$NVOL3.*.* )
+> CATALOG \NEWSYS.$NVOL.DB FOR \NEWSYS.*.*.* NO LISTALL;
```

- This example duplicates all tables, collations, and files, but no indexes or views, that reside on subvolume \$VOL1.PERSNL to subvolume \$VOL1.NWPERS:

```
>> DUP $VOL1.PERSNL.*, *.NWPERS.*, INDEXES OFF, VIEWS OFF
+> NO LISTALL;
```

- The PERSNL subvolume contains the EMPLOYEE, DEPT, and JOB tables and the EMPLIST and MGRLIST views. This commands duplicate all tables and their indexes, but only the MGRLIST view:

```
>> DUP $VOL1.PERSNL.*, *.NWPERS.*, VIEWS OFF NO LISTALL;
>> DUP $VOL1.PERSNL.M*, $VOL1.NWPERS.*, VIEWS EXPLICIT
+> NO LISTALL;
```

If other files or tables on the PERSNL subvolume have names that also begin with the letter M, you must provide a more specific source fileset list in the second command: for example, \$VOL1.PERSNL.MGRL* or \$VOL1.PERSNL.MGRLIST.

Dynamic SQL

Dynamic SQL is a form of embedded SQL that allows you to build, compile, and execute SQL DCL, DDL, and DML statements during program execution. You can use dynamic SQL in programs that build SQL statements at execution time or that process SQL statements entered by users or generated by applications on personal computers.

Two dynamic SQL statements, PREPARE and EXECUTE, can also be used outside embedded SQL programs to eliminate the need to recompile SQL statements that you execute multiple times in a single SQLCI session.

The below table summarizes dynamic SQL statements:

Summary of Dynamic SQL Statements

DESCRIBE	Returns information about output variables of prepared statements
DESCRIBE INPUT	Returns information about input parameters of prepared statements
EXECUTE	Executes a compiled statement
EXECUTE IMMEDIATE	Executes an SQL statement contained in a host variable
PREPARE	Compiles a DDL, DML, or DCL statement for later execution by EXECUTE
RELEASE	Deallocates memory for a dynamic SQL statement referred to through a host variable

Determining When to Use Dynamic SQL

Dynamic SQL can be less efficient than static SQL because more work is deferred until run time. If you do not know the whole text of an SQL statement at development time, but there are only a few alternatives, you might want to program the alternatives into your application.

If your application requires greater flexibility, dynamic SQL can be useful. For example, you could use dynamic SQL if your application requires:

- Flexibility to construct SQL statements at run time: for example, an interactive interface that is similar to SQLCI, but is designed for an inexperienced user.
- Restriction of access to data in a table: for example, the program might code an UPDATE statement for certain columns in a table, but allow the user to enter any selection criteria (WHERE clause) at run time.
- Client-server support with deferral of definition of SQL statements until run time: for example, when the user of an application on a personal computer wants to manipulate data in a NonStop SQL/MP database on a host system. Such a program cannot use SQLCI. The user formulates an SQL statement on the personal computer and the application sends it to a server process on the system over Multilan or another communications protocol.

If you plan to execute a dynamic SQL statement only once, you can use EXECUTE IMMEDIATE to execute the statement, and save any memory that would have stored the execution plan.

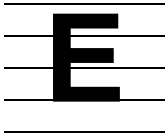
Features of Dynamic SQL

When you write a program that uses dynamic SQL, you use many of the same SQL statements as you would in static SQL. You can perform most of the same operations using dynamic SQL statements that you perform with static SQL statements. You can use DDL, DML, and DCL statements in both modes.

The difference is that all or part of a dynamic SQL statement is obtained at run time from the user, or generated by your program. Your program stores the statement in a character host variable, and then compiles and executes it. With dynamic SQL statements, you must perform some additional operations (such as building descriptors for host variables) that are performed for you when you use static SQL statements.

After compilation, SQL executes statements in the same way, whether they are dynamic or static. SQL places the results of dynamic SQL statements into output variables; you can use DESCRIBE to obtain information about those variables.

For information about dynamic SQL, see the SQL/MP programming manual for your host languages.



EDIT Command

EDIT is an SQLCI command that invokes the EDIT text editor.

```
EDIT [ [ file [!] ]  
      [ " [ file [!] ] [ ; edit-cmd ] ... " ] ;  
      [ [ file [!] ] " [ ; edit-cmd ] ... " ]
```

file

is a Guardian name that specifies the file to EDIT.

[!]

directs EDIT to create *file* if it does not already exist.

edit-cmd

is a text editor command. See the *EDIT User's Guide and Reference Manual*.

Note that you must use a form of the command that includes quotation marks if you specify *edit-cmd*. If you do not, SQLCI interprets the first semicolon on the line as the end of the SQLCI statement, not as input to the editor.

Examples—EDIT

- This example starts an EDIT session within an SQLCI session to edit the file FINDEMP:

```
>> EDIT FINDEMP;  
TEXT EDITOR - T9601D20 - (01JUN93)  
CURRENT FILE IS $VOL1.PERSNL.FINDEMP  
*
```

- This example passes commands to EDIT and starts an EDIT session within an SQLCI session (EDIT output is not shown):

```
>> EDIT MYDATA" ;L/1994/ ;XVS" ;
```

Embedded SQL

Embedded SQL, or programmatic SQL, is the application programming interface for SQL. It consists of a set of SQL statements and declarations you can include in programs written in C, COBOL, Pascal, or TAL.

The language in which you write an embedded SQL program is called the *host* language or host programming language in SQL/MP documentation. A separate SQL/MP programming manual exists for each of the four host languages supported by NonStop SQL/MP. From each language you can also use the basic SQL statements (but not the SQLCI commands) documented in this manual.

You can write embedded SQL programs that run as OSS processes or Guardian processes in the host language C. Embedded SQL programs written in COBOL, Pascal, or TAL must run as Guardian processes.

Embedded SQL programs communicate with SQL through host language variables declared in an SQL declare section within the host language declare section and through special SQL data structures called SQLCA, SQLDA, and SQLSA. Embedded SQL programs declare and use cursors to process statements that return more than one row of data; they can also use the SQL directive WHENEVER to test for exception conditions.

An embedded SQL statement can be either static SQL (a statement coded directly into the source code and compiled before the program execution) or dynamic SQL (a statement built and compiled during the program execution).

For more information, see these entries:

[Dynamic SQL](#)

[INCLUDE SQLCA
Directive](#)

[DECLARE CURSOR
Statement](#)

[Static SQL](#)

[INCLUDE SQLDA
Directive](#)

[OPEN Statement](#)

[BEGIN DECLARE
SECTION Directive](#)

[INCLUDE SQLSA
Directive](#)

[FETCH Statement](#)

[END DECLARE
SECTION Directive](#)

[WHENEVER DIRECTIVE](#)

Also, see the SQL/MP programming manual for the host programming language.

END DECLARE SECTION Directive

END DECLARE SECTION is a host program directive that ends a host program Declare Section for declaring host variables to use in SQL statements.

For more information about declaring host variables, see the SQL/MP programming manual for your host language.

```
END DECLARE SECTION
```

Example—END DECLARE SECTION

This statements from a C, Pascal, or TAL program declare host variables that correspond to the columns of the table PARTS:

```
EXEC SQL BEGIN DECLARE SECTION;
      EXEC SQL INVOKE PARTS AS PARTSREC;
EXEC SQL END DECLARE SECTION;
```

ENV Command

ENV is an SQLCI command that displays attributes of the current SQLCI session. You can use ENV at either the SQLCI prompt (>>) or the select-in-progress prompt (S>).

```
ENV ;
```

Considerations—ENV

- These are the fields in the ENV display:

CATALOG	The current default catalog.
LANGUAGE	The language of text in the message file.
LOG	The log file for the SQLCI session.
MESSAGEFILE	The current SQL message file.
MESSAGEFILE VRSN	The version of the SQL/MP software in use.
OUT	The OUT file for the SQLCI session.
OUT_REPORT	The OUT_REPORT file for the SQLCI session.
SYSTEM	The current default node.
TRANSACTION ID	The transaction identifier of the current TMF transaction if a transaction is in progress.
VOLUME	The current default volume.
WORK	The TMF transaction status (IN PROGRESS or NOT IN PROGRESS).

- You can use the CATALOG, LOG, OUT_REPORT, SYSTEM, and VOLUME commands to change current default values for the SQLCI session. You can change the OUT file with the OUT command (or with the OUT run option when you start an SQLCI session).

You can specify a different SQL message file for an SQLCI session by setting the `=_SQL_MSG_node` DEFINE before you start the SQLCI session. Changing the DEFINE after you start SQLCI, however, does not change the message file. (You cannot change the message file version and language directly because they depend upon the message file in use.)

Example—ENV

This example shows the output of an ENV command that is entered at a select-in-progress prompt.

```
S> ENV;
-----
Current Environment
-----
CATALOG          \SYS1.$VOL1.SALES
LANGUAGE         AMERICAN ENGLISH
LOG              $VOL1.SUBVOL1.LOGFILE
MESSAGEFILE      \SYS.$SYSTEM.SYSTEM.SQLMSG
MESSAGEFILE VRSN 315
OUT              \SYS1.$TERM1
OUT_REPORT       $$.#PRINTER
SYSTEM           \SYS1
TRANSACTION ID   \SYS1.0.474330
VOLUME           $VOL1.PERSNL
WORK             IN PROGRESS
S>
```

ERROR Command

ERROR is an SQLCI command that displays the error text associated with an error number and, optionally, the cause and effect of the error and suggestions for recovery.

```
ERROR [ [ type ] number [ , { DETAIL } ] ] ;
      [ [ { BRIEF } ] ] ]
```

type is:

```
{ AUD }
{ DP  }
{ FS  }
{ OS  }
{ SIO }
{ SORT }
{ SQL }
```

type

specifies the type of error and can be one of these:

AUD	Audit-fixup process
DP	Disk process
FS	File system
OS	Guardian operating system
SIO	Sequential I/O
SQL	SQL command
SORT	Sort

If you omit *type*, SQLCI displays the error text for all operations that generate the specified error number.

number

is a positive or negative number that identifies the error you want described. If you omit *number*, SQLCI displays information about the most recent error. The `DISPLAY_ERROR` and `WARNINGS` session options determine which errors are reported. For more information, see [SET SESSION Command](#) on page S-39.

`DETAIL` or `BRIEF`

specifies the error information to display.

`DETAIL` Display error text, cause, and effect, plus suggestions for recovery.

`BRIEF` Display error text only.

The default is `DETAIL` unless you have previously set the `ERROR_TEXT` session option to `BRIEF`. For more information, see [SET SESSION Command](#) on page S-39.

Examples—ERROR

- This example displays the error text of file-system error number 1066:

```
>> ERROR FS 1066, BRIEF;
Internal error: Occurred in OPEN.
>>
```

- This example displays the text, cause, and effect of SQL error number 1249:

```
>> ERROR SQL 1249, DETAIL;
A column cannot be added to an entry-sequenced table.
Cause:
Stated in the error message.

Effect:
The statement fails.
```

Recovery:

Create a new table with the correct number of columns that has the same contents as the old table.

>>

Error Messages

SQL returns error and warning information through SQLCI and through programmatic interfaces.

Each SQL error message is associated with a negative number and each SQL warning message is associated with a positive number. Some SQL messages are associated with both a negative (error) and a positive (warning) number because the problem can cause an error in certain situations and a warning in others.

You can use the ERROR command in SQLCI to get information about a specific error or warning. You can also modify the SQLCI session options DISPLAY_ERROR, ERROR_ABORT, ERROR_TEXT, and WARNINGS to control SQLCI action when an error or warning occurs. For more information, see [ERROR Command](#) on page E-4 or [SET SESSION Command](#) on page S-39.

Within a program, SQL returns error information to the SQLCODE field in the SQLCA data structure. Specific error messages are not intended to be handled programmatically, but are intended to be passed directly to users or stored in tables. There are cases in which an SQL statement can return a different error than it did in a previous version of SQL.

For more information about handling errors and warnings within programs, see the SQL/MP programming manual for your host language.

For a complete list of SQL error and warning messages, including those that are displayed by SQLCI or sent to programs in addition to those sent to the EMS subsystem, see the *SQL/MP Messages Manual*.

EXECUTE Statement

EXECUTE is a dynamic SQL or SQLCI statement that executes an SQL statement previously compiled by the PREPARE statement.

```
EXECUTE { stmt-name
        { :stmt-variable }

[ USING [?param=] value [, [?param=] value ]... ]
[ USING :variable [ , :variable ] ... ]
[ USING DESCRIPTOR :in-sqlda ]

[ RETURNING { :variable
              { USING DESCRIPTOR :out-sqlda } } ]

value is:
```

```
{ literal
  CURRENT_TIMESTAMP
  COMPUTE_TIMESTAMP }
```

```
{ stmt-name }
```

```
{ :stmt-variable }
```

specifies the compiled statement to execute by the name assigned to it in the PREPARE statement.

stmt-name is the name. Use this form to specify the name in SQLCI or in programs.

:stmt-variable is a host variable of SQL type CHAR or VARCHAR that contains the name. Use this form to specify the name only in programs.

```
[ USING [ ?param=] value [, [?param =] value ] ... ]
[ USING :variable [, :variable ] ... ]
[ USING DESCRIPTOR :in-sqlda ]
```

specifies values for parameters in the compiled statement. Use the first form in SQLCI, the second form in a program that has information about the parameters, and the third form in a program that uses DESCRIBE INPUT to dynamically retrieve information about the parameters.

Whatever form you use, these rules apply:

- You must supply a value for each parameter in the statement to be executed that does not currently have a value, including all unnamed parameters.
- The data type of a parameter value must be compatible with the data type of the associated parameter.

- Unnamed parameter values are substituted for parameters in the SQL statement by position. The *i*-th value in the USING clause or in the SQLDA is the value for the *i*-th formal parameter.
- Any parameter values you specify in the USING clause override values you previously specified in SET PARAM commands, but only for this execution of the statement.

param

(used in SQLCI) is the name of a parameter to be assigned the value that immediately follows. If you specify the same parameter-value pair more than once, SQL uses the last specification.

To assign values to unnamed parameters, omit *?param=* and specify the values in the same order that the unnamed parameters appear in the prepared command.

value

(used in SQLCI) is a value for a parameter. A *value* can be one of:

- A numeric or string literal, optionally enclosed in quotation marks
- CURRENT_TIMESTAMP—an SQLCI function that returns a Julian timestamp for the current date and time as a value of data type NUMERIC 18 or LARGEINT.

SQL evaluates CURRENT_TIMESTAMP when the statement in which it appears executes. As a result, CURRENT_TIMESTAMP in an EXECUTE statement returns the time that the EXECUTE executes; but CURRENT_TIMESTAMP in a SET PARAM returns the time that the SET PARAM executes, not the time that an EXECUTE that uses the parameter executes.

- COMPUTE_TIMESTAMP (*date*)—an SQLCI function that returns a Julian timestamp for the date and time you specify in *date* as a value of data type NUMERIC 18 or LARGEINT.

For more information about the functions, see [CURRENT_TIMESTAMP Function](#) on page C-174 or [COMPUTE_TIMESTAMP Function](#) on page C-62. For examples of their use with EXECUTE, see the examples later in this entry.

:variable

(used in programs) is a host variable that contains a value for a parameter in the statement.

:in-sqllda

(used in programs) is an SQLDA filled by DESCRIBE INPUT that points to values for parameters in the statement.

```
RETURNING { :variable
            { USING DESCRIPTOR :out-sqllda } }
```

(used only if a dynamic INSERT RETURNING is executed) directs SQL to return the SYSKEY for the last record inserted.

:variable is a host variable in which to return the key. It must be of an appropriate type for the key (INTEGER UNSIGNED for tables with entry-sequenced or relative organization; LARGEINT SIGNED for tables with key-sequenced organization).

:out-sqllda is an SQLDA set by the DESCRIBE statement that tells where to return the key. If you use this option, your program must have set the VAR_PTR field in the SQLDA to point to a buffer to receive the SYSKEY value.

If you use the RETURNING clause for a table with a clustering key, SQL returns only the appended SYSKEY. To find the record, specify the clustering key columns and the SYSKEY column (the most efficient method), or only the SYSKEY column.

Considerations—EXECUTE

- A statement must be compiled by PREPARE before you can EXECUTE it but after it is compiled, you can EXECUTE the statement multiple times without recompiling it.

Host language scoping rules apply to EXECUTE in programs. For more information, see the SQL/MP programming manual for the host language you use.

- You must supply a value for each formal parameter in the statement to be executed, and each value must be of a type compatible with the associated formal parameter.

You can specify parameter values for named formal parameters with the USING clause, with the SQLCI SET PARAM command, or with the TACL PARAM command. (A TACL PARAM named “A” is the same as a parameter named “?A” in SQLCI.) You can specify parameter values for unnamed parameters only with the USING clause.

- Unless a CONTROL QUERY BIND NAMES AT EXECUTION directive is in effect when a PREPARE executes, the compiled statement uses the defaults and DEFINES in effect at the time it is prepared, not the time it executes. For information, see [CONTROL QUERY Directive](#) on page C-74 or [Name Resolution](#) on page N-2.
- You cannot execute a prepared DDL statement (except UPDATE STATISTICS) that operates on nonaudited tables or indexes within a user-defined TMF transaction.

Examples—EXECUTE

- This example uses PREPARE and EXECUTE to compile the statement stored in the variable :DYNSTMT and execute it using parameter values stored in the variables :PARTNUM, :PRICE, and :PARTDESC:

```
EXEC SQL
  PREPARE OPERATION FROM :DYNSTMT;
EXEC SQL
  EXECUTE OPERATION USING :PARTNUM, :PRICE, :PARTDESC;
```

- This SQLCI example uses PREPARE to compile a statement once, then executes the statement multiple times with different parameter values:

```
PREPARE FINDEMP FROM "SELECT * FROM PERSONL.EMPLOYEE"
  &"WHERE SALARY > ? AND JOBCODE = ? ";
EXECUTE FINDEMP USING 30000, 200;
EXECUTE FINDEMP USING 40000, 100;
```

- This SQLCI statements use CURRENT_TIMESTAMP and COMPUTE_TIMESTAMP in EXECUTE USING clauses as values for both LARGEINT and TIMESTAMP fields. Notice how the prepared statement uses parameter ?T directly for the type LARGEINT field but converts it for the type TIMESTAMP field.

```
CREATE TABLE DEMO (DEMOKEY NUMERIC,
  JDATE LARGEINT, DDATE TIMESTAMP, PRIMARY KEY DEMOKEY);
PREPARE MYSTMT FROM
  "INSERT INTO DEMO VALUES (?K, ?T, CONVERTTIMESTAMP(?T) )";
EXECUTE MYSTMT USING ?K=1, ?T=CURRENT_TIMESTAMP;
EXECUTE MYSTMT USING ?K=2, ?T=COMPUTE_TIMESTAMP (10/13/93);
```

- This SQLCI statements set parameter values with the SET PARAM command but override one of the parameter values with a value in the USING clause of the EXECUTE statement:

```
VOLUME PERSONL;
PREPARE NEWJOB FROM "INSERT INTO JOB VALUES (?CODE, ?DESC)";
SET PARAM ?CODE 950, ?DESC "TECHNICIAN";
EXECUTE NEWJOB USING ?DESC = "SR. TECHNICIAN";
```

- This SQLCI statements use both SET PARAM and the USING clause of the EXECUTE statement to supply parameters for a prepared SELECT.

Notice that the first two EXECUTE statements use the part number from SET PARAM and use positional notation to provide a supplier number (the first parameter) on the EXECUTE. The third EXECUTE uses both positional and named notation to supply parameter values. The fourth EXECUTE uses the ?PNUM value from the SET PARAM (which was overridden, but not changed by the using clause in the third EXECUTE).

```
PREPARE FINDSUP FROM "SELECT * FROM INVENT.PARTSUPP"
  &" WHERE PARTNUM = ?PNUM AND SUPPNUM = ?";
SET PARAM ?PNUM 4103;
EXECUTE FINDSUP USING 6;
```



```
EXECUTE FINDSUP USING 25;
EXECUTE FINDSUP USING 6, ?PNUM = 5504;
EXECUTE FINDSUP USING 8;
```

EXECUTE IMMEDIATE Statement

EXECUTE IMMEDIATE is a dynamic SQL statement used in a host program to compile and execute an SQL statement whose text is contained in a host variable.

```
EXECUTE IMMEDIATE :host-var
```

:*host-var*

identifies a host variable declared as an alphabetic or alphanumeric data item; *host-var* must contain the SQL statement as a string literal.

If the SQL statement is an INSERT statement, the statement cannot contain the RETURNING clause.

The SQL statement must not contain parameters or refer to host variables.

Considerations—EXECUTE IMMEDIATE

- You can use EXECUTE IMMEDIATE for any DDL, DML, or DCL, or DSL statement, except OPEN, CLOSE, and SELECT. (Use a cursor to process a SELECT statement.)
- If the program declares an SQLSA, EXECUTE IMMEDIATE does not return execution statistics as described under INCLUDE SQLSA. EXECUTE IMMEDIATE does not return compilation statistics.

Example—EXECUTE IMMEDIATE

This SQL statement from a C program executes an SQL statement whose text is contained in the host variable named :statement:

```
EXEC SQL EXECUTE IMMEDIATE :statement;
```

EXISTS Predicate

EXISTS is a predicate that determines whether any rows satisfy conditions in a subquery. The EXISTS predicate evaluates to true if the subquery finds at least one row that satisfies the search condition.

[NOT] EXISTS *subquery*

In an EXISTS predicate, the result of *subquery* can be a table of more than one column.

An EXISTS subquery is typically correlated with an outer query.

Examples—EXISTS

- This query searches for departments that have no engineers (job code 420):

```
SELECT DEPTNAME FROM PERSONL.DEPT D
WHERE NOT EXISTS
  (SELECT JOBCODE
   FROM PERSONL.EMPLOYEE E
   WHERE E.DEPTNUM = D.DEPTNUM
   AND JOBCODE = 420);
```

- This query searches for parts with less than 20 units in the inventory:

```
SELECT PARTNUM, SUPPNUM
FROM INVENT.PARTSUPP PS
WHERE EXISTS
  (SELECT PARTNUM
   FROM INVENT.PARTLOC PL
   WHERE PS.PARTNUM = PL.PARTNUM
   AND QTY_ON_HAND < 20);
```

- This query finds the locations of salespersons (employees with jobcode 300).

The EXISTS predicate contains a subquery that determines which locations have salespersons. The subquery depends on the value of DEPT.DEPTNUM from the outer query. In this case, the subquery must be evaluated for each row of the result table where DEPT.DEPTNUM equals EMPLOYEE.DEPTNUM. Column DEPT.DEPTNUM is an example of using an implicit correlation name as an outer reference.

```
SELECT DEPTNUM, LOCATION
FROM DEPT
WHERE EXISTS (SELECT JOBCODE
              FROM EMPLOYEE
              WHERE DEPT.DEPTNUM = EMPLOYEE.DEPTNUM
              AND JOBCODE = 300);
```

EXIT Command

EXIT is an SQLCI command that ends an SQLCI session. Pressing Ctrl-Y is the same as typing EXIT.

```
E[EXIT] [ ; ]
```

Control returns to the process from which you started SQLCI, usually the command interpreter.

If a user-defined transaction is in progress, SQLCI prompts you to specify whether you want to commit or roll back the transaction.

Example—EXIT

In this example, the EXIT command is abbreviated to E:

```
>> E
```

```
End of SQLCI session
```

EXPLAIN Directive

EXPLAIN is a directive or SQL utility that describes the execution plans for queries.

You can execute EXPLAIN through SQLCI (as described here) or by using an option on the SQLCOMP command line (as described in the SQL/MP programming manual for your host language).

```
EXPLAIN [ PLAN FOR ] { statement
                       { statement-name }
```

PLAN FOR

is an optional clause that does not affect the EXPLAIN output.

statement

is an SQL DML statement, by itself or enclosed in single or double quotation marks.

statement-name

is the name of a prepared SQL statement.

Considerations—EXPLAIN

- You can use the information in an EXPLAIN report to tune queries and to help determine whether to add or drop indexes for a database. For listings of sample

EXPLAIN reports and for a detailed explanation of what to look for when you analyze EXPLAIN output, see the *SQL/7MP Query Guide*.

Note that an EXPLAIN report is based on information available at the time you generate the report. If access paths or statistics change before you execute a query for which you obtained an EXPLAIN report, SQL might use a different execution plan. For example, the EXPLAIN execution plan for a SELECT statement reflects the access paths that exist at the time. If you use EXPLAIN to generate an execution plan for the same statement after dropping an index used in the original plan, the new plan will be different.

- EXPLAIN reports generated through the SQLCOMP command rather than through SQLCI include an optional section that lists each DEFINE used in an SQL statement within the program and the Guardian name associated with that DEFINE at compilation time. This portion of the report is generated in the form of ADD DEFINE commands or INFO DEFINE output, depending upon the option you select.

For more information, including samples of such reports, see the SQL/MP programming manual for the host language you use.

- The execution plan for each DML statement described in an EXPLAIN report is divided into one or more steps: one for a scan of each table in the FROM clause and one for each union operator in the query. Each step involves one or more of these operations:
 - Scan of a table
 - Join of two or more tables
 - Insert into a table
 - Sort operation
 - Hash operation

The plan shows different types of information for each type of operation. For joins, for example, the plan shows the order of joining, join methods, and sorting operations.

Predicate information in an execution plan can contain multibyte characters. If multibyte characters are present, those characters might not be displayable on your output device.

This example shows the format of an execution plan. Optional lines and clauses in the report are shown in brackets or braces, as for syntax notation. This figure is an alphabetic list of the elements of the plan with an explanation of each element.

```

Query plan <n>  [ : Will utilize parallel execution ]
SQL request    : { Delete | Update | Select | Insert |
                  Insert-Select | Union of Selects      }
-----
Plan step n [ : Perform an [ [Inner|Left] Join|Union ] ]
[ Join strategy : [Nested | [Hybrid] Hash | ] ]

```

```

[           [Key Sequenced] Merge] Join           ]
[ (This section appears only for parallel         ]
[   plans and is described under "Parallel Execution ]
[   Plan" in the text that follows the figure.     ]
[   Typical lines are shown below, though the details ]
[   of section contents vary depending on the plan.) ]
[ ]
[ Each operation is performed in parallel for this step. ]
[ ]
[ Each ESP [ from previous step ] will read one of ... ]
[   partition partition partition ... ]
[ The ESPs will be started in the CPU's numbered ]
[   n n n ... ]
[ Each ESP will perform a [ Hybrid ] Hash Join | ]
[   [ [Key Sequenced] Merge Join ... ] ]
[ ]
[ Plan Forced      :{Join sequence          } forced by user ... ]
[                   :{Join method           } ]
[                   :{Join sequence         } ]
[                   and join method } ]
Characteristic :
Operation n      :{ Insert                      }
                  :{ Scan                        }
                  :{ Union of plan step n ... n }
Table           :
[ Accessed via view : ] (Appears for a protection view)
Access type     :
Lock mode       :
Column processing :
  Access path n :
  SBB for reads  :
[ Begin key pred. : ]
[ End key pred.   : ]
[ MDAM predicate set: ]
[   next set:    ]
Index selectivity :
Index pred.       :
Base table pred.  :
[ Type of [ Update | Delete ] : ]
[ SBB for [ Insert | Update ] : ]
[ Seq Blocksplitt : ]
Executor pred.    :
[ Executor aggr.  : ]
[ Pred. selectivity : ]
[ DP2 aggr.       : ]
Table selectivity :
Expected rowcount :
Operation cost    :
-----
Operation n      : Sort (Appears for sort operations)
Requested        :
Sort rows in the :
Purpose          :
Sort technique   :
Sort type        :
UPS workspace    :

```

```

Sort key columns :
[ Expected rowcount : ] (Appears if sort is for GROUP BY)
Sort cost       :
-----
Operation n      : Hash (Appears for hash operations)
Requested        :
Hash rows in the :
Purpose          :
Hash key columns :
Hash cost        :
Total cost       :
-----

```

Table E-1. EXPLAIN Plan Header Lines (page 1 of 5)

Item	Action
Access path	<p>Identifies the path used for retrieving rows from the base table. The path can be a primary path (the base table) or an alternate path (an index defined on the base table).</p> <p>The phrase <i>path forced</i> indicates that the path was specified by the user with a CONTROL TABLE directive.</p> <p>Access path also states whether the table is partitioned and whether access is sequential and pages are kept in cache as long as possible (called sequential cache). If multiple indexes are scanned (because of OR'ed predicates in the WHERE clause), more than one access path can be shown.</p>
Access type	Specifies the unit of locking (record or table) and the access method that controls lock requests (STABLE, REPEATABLE, or BROWSE).
Accessed via view	Specifies the fully qualified Guardian name for a protection view whose underlying table is specified in the report line beginning with "Table." For a shorthand view, the view definition becomes part of the query so that the query plan describes operations on the underlying tables or views.
Base table pred.	Shows the predicates evaluated on the base table and indicates that evaluation is performed by the disk process.
Begin key pred.	Specifies the predicates used to position to the first row to scan, including collations explicitly (but not implicitly) used in the predicates.
Characteristic	Describes special characteristics of the step, such as whether it executes once before the operation or once per row retrieved. For joins, indicates which previous step produced the rows to be joined.
Column processing	Indicates the number of columns to be retrieved and the total number of columns in the tables or views from which they are retrieved.
DP2 aggr.	Indicates that aggregate functions are evaluated by DP2.
End key pred.	Specifies the predicates used to position to the last row to scan, including collations explicitly used in the predicates.

Table E-1. EXPLAIN Plan Header Lines (page 2 of 5)

Item	Action
Column processing	Indicates the number of columns to be retrieved and the total number of columns in the tables or views from which they are retrieved.
DP2 aggr.	Indicates that aggregate functions are evaluated by DP2.
End key pred.	Specifies the predicates used to position to the last row to scan, including collations explicitly used in the predicates.
Executor aggr.	Indicates that aggregate functions are evaluated by the executor and shows whether they are derived from the first row returned by the scan or computed for each group.
Executor pred.	Shows predicates evaluated by the executor and describes when they are evaluated: <ul style="list-style-type: none"> ● On rows retrieved by the scan ● On rows after sorting ● On null augmented rows ● From the HAVING clause
Expected rowcount	For an operation other than a sort, shows the number of rows the optimizer expects to be returned. For a join query, the rowcount is cumulative for each join operation. Appears for sort operations only if the purpose of the sort is to group rows for a GROUP BY clause. In this case, shows the number of rows expected after the GROUP BY operation finishes.
Hash cost	Indicates the relative cost of the hash operation. Lower numbers indicate more efficient and less costly execution.
Hash key columns	Lists hash key columns ordered by selectivity, with the most selective column first.
Hash rows in the	Specifies whether the rows hashed are from the scan from the previous operation (current table) or from a composite table.
Index pred.	Shows the predicates evaluated on the index and indicates whether evaluation is performed by the file system or the disk process.
Index selectivity	Shows the percentage of the index that is scanned for the operation.
Join strategy	Shows the type of join used: nested, (sort) merge, key-sequenced merge, hash, or hybrid hash. For parallel or repartitioned hash joins, additional lines list the partitions accessed by the ESPs and the processors for the ESPs.
Lock mode	Indicates whether locks are exclusive, shared, or the default. With the default in effect, the system determines the actual lock mode used.
MDAM predicate set	Indicates the predicate set used for the first step in the query.
next set	Indicates the predicate set used for a subsequent step in the query.
Operation <i>n</i>	Indicates the type of operation and the order of the operation in the overall plan.

Table E-1. EXPLAIN Plan Header Lines (page 3 of 5)

Item	Action
Operation cost	Indicates the relative cost of performing the operation. Lower numbers indicate more efficient and less costly operation.
Parallel Execution Plan	<p>Describes the parallel execution plan for the operation, including:</p> <ul style="list-style-type: none"> ● The number of executor server processes (ESPs) that perform each operation in parallel ● The name of the system and the processors in which the ESPs are started ● The names of the volumes on which partitions of the base table reside ● The names of the volumes on which rows are redistributed to promote parallel access ● The number and purpose of each sort started by each ESP in parallel. <p>If the query requires a sort, the plan shows the number of sorts performed in parallel.</p> <p>For a join query, describes the plan as one of these three types:</p> <ul style="list-style-type: none"> ● Matching partitions (nested join only) ● Parallel access ● Repartitioned
Plan Forced	Indicates that the join method, the join sequence, or both were forced by the user with a CONTROL TABLE directive.
Plan step	<p>Indicates the number of the plan step.</p> <p>Specifies whether the plan step involves a union or join operation. If the plan step involves a join operation, specifies the type of join (inner or left) and the join strategy: nested, (sort) merge, key-sequenced merge, or hash.</p>
Pred. selectivity	Specifies the percentage of the table or index that is evaluated to test the search conditions of the predicates.
Purpose	<p>Specifies why the sort or hash join is required:</p> <ul style="list-style-type: none"> ● To order rows before the join phase ● To order rows for an ORDER BY ● To form groups of rows for a GROUP BY ● To compute an aggregate DISTINCT ● To discard duplicates for a DISTINCT
Plan step	<p>Indicates the number of the plan step.</p> <p>Specifies whether the plan step involves a union or join operation. If the plan step involves a join operation, specifies the type of join (inner or left) and the join strategy: nested, (sort) merge, key-sequenced merge, or hash.</p>

Table E-1. EXPLAIN Plan Header Lines (page 4 of 5)

Item	Action
Query plan <i>n</i>	Specifies the number of the query and whether the plan involves parallel execution.
Requested	Specifies whether the sort or hash operation was requested by the optimizer or explicitly requested by the user.
SBB for [Insert] [Update]	Specifies whether virtual sequential block buffering (VSBB) is used for insert or update operations.
SBB for reads	Specifies whether real or virtual sequential block buffering (SBB) is used for read operations.
Seq Blocksplrit	Indicates that the user requested blocks to be split as for sequential inserts by specifying the CONTROL TABLE SEQUENTIAL BLOCKSPLIT ON option.
Sort cost	Indicates the relative cost of the sort operation. Lower numbers indicate more efficient and less costly execution.
Sort key columns	Identifies columns used in the sort by column names qualified by correlation or table names or (if the sort column is an expression) by position in the select list. Also lists collations associated with the sort columns.
Sort rows in the	Specifies whether the rows sorted are from the result of a SELECT statement or a union of SELECT statements, on the current table, or on a composite table.
Sort technique	Describes the sort technique: FastSort or insertion into a key-sequenced file.
Sort type	Describes the type of sort: insertion into an entry-sequenced disk file, parallel sorting, parallel sort/merge, Sortmerge, or User Process Sort.
SQL request	Specifies the type of statement that is the subject of the plan: DELETE INSERT-SELECT UPDATE UNION of SELECT SELECT UNION of INSERT-SELECTs INSERT
Table	Specifies the fully qualified Guardian name of the table. For scan or join operations, the name represents the table being accessed; for an INSERT operation, the name represents the table to receive the inserted data.
Table selectivity	The percentage of the table that is reflected in the result.

Table E-1. EXPLAIN Plan Header Lines (page 5 of 5)

Item	Action
Total cost	<p>The total cost of executing the statement. Cost is a relative measure of the resources needed. Lower numbers indicate less costly execution.</p> <p>This measure is useful only for comparing different ways of specifying the same query (for example, using a join instead of a subquery). Total Cost cannot not be used to compare the efficiency of executing different queries.</p>
Type of [DELETE] [UPDATE]	<p>Indicates the type of DELETE or UPDATE operations: cursor, subset, or unique.</p> <p>For a cursor DELETE or UPDATE, the executor opens an internal cursor to read the qualified rows and performs the requested operation for each such row.</p> <p>For a subset DELETE or UPDATE, the file system positions on the first qualifying row (if the row has been determined) or on the first row of the table. The file system sends a message to the disk process to perform the operation on all rows that qualify. The file system might have to send multiple requests because the disk process performs a limited amount of work in a single request.</p> <p>For a unique DELETE or UPDATE, the disk process performs the operation on the specific row. Exactly one row is deleted or updated.</p>
UPS workspace	Indicates the size of the workspace needed for the User Process Sort operation.

Examples—EXPLAIN

- These examples shows four different ways to request an EXPLAIN report for the same simple query:

Example 1.

```
>> EXPLAIN SELECT * FROM MYTABLE;
```

Example 2.

```
>> EXPLAIN "SELECT * FROM MYTABLE";
```

Example 3.

```
>> EXPLAIN 'SELECT * FROM MYTABLE';
```

Example 4.

```
>> PREPARE MYQUERY FROM SELECT * FROM MYTABLE;
. .
--- SQL command prepared.
>> EXPLAIN MYQUERY;
```

- These examples show three different ways to request an EXPLAIN report for a query that uses multiple lines in SQLCL:

Example 1.

```
>>EXPLAIN SELECT U1,U2, MIN(U3), MAX(U3)
      +>FROM \SYS.$VOL.CAT.T
      +>GROUP BY 1 , 2;
```

Example 2.

```
>>EXPLAIN " SELECT U1,U2, MIN(U3), MAX(U3) "
      +>&" FROM \SYS.$VOL.CAT.T "
      +>&" GROUP BY 1 , 2";
```

Example 3.

```
>>PREPARE SAVQ FROM SELECT U1,U2, MIN(U3), MAX(U3)
      +>FROM \SYS.$VOL.CAT.T
      +>GROUP BY 1 , 2;
      . .
      --- SQL command prepared.
>> EXPLAIN SAVQ;
```

- For examples of EXPLAIN reports, see the *SQL/MP Query Guide*. For an example of a DEFINE report (an optional portion of an EXPLAIN report that describes the DEFINES used in SQL statements within a program), see the SQL/MP programming manual for your host language.

Expressions

An expression specifies a value. An expression can be a simple character string, a numeric literal, a column name, or a condition (CASE expression) that specifies the value of the column in a row of a table. An expression can also include function calls and arithmetic operators. All these are simple expressions:

"ABILENE"	A character string
-57	A numeric literal
CUSTNAME	The value in column CUSTNAME
COUNT (DISTINCT CITY)	The COUNT function applied to the values in column CITY

An expression can have a character, numeric, date-time, or INTERVAL data type. The data type of an expression is the data type of the value of the expression.

The remainder of this entry discusses numeric, date-time, and INTERVAL expressions. For more information about character expressions, see [Character Expressions](#) on page C-14. For more information about conditional expressions, see [CASE Expression](#) on page C-2.

Numeric, Date-Time, and Interval Expressions

A numeric, date-time, or INTERVAL expression consists of one or more numeric, date-time, or INTERVAL operands connected by arithmetic operators, as shown in this diagram.

```
operand [ [ arith-operator operand ] ... ]
```

operand is:

```
{
  [+] numeric-operand [ UNITS field ]
  [-]
  {
    date-time-operand
    [ [ start-field TO ] end-field ]
    [ UNITS field ]
  }
  [+] interval-operand
  [-]
  {
    [ start-field [(digits)] [ TO end-field ] ]
    [ UNITS field ]
  }
}
```

[+] is a unary operator

[-]

arith-operator is:

```
{
  **
  *
  /
  +
  -
}
```

numeric-operand

is a column name (possibly qualified by a correlation name), a literal, a function invocation, a host variable, a parameter name, or an expression that evaluates to a numeric value.

UNITS field

used after a numeric operand, converts the value of the operand to a value of type *INTERVAL field*; used after a date-time or INTERVAL operand, converts the value of the operand to a numeric value that represents the number of units of field-type *field* that are contained in the value.

date-time-operand

is a column name (possibly qualified by a correlation name), a literal, a function invocation, a host variable, a parameter name, or an expression that evaluates to a date-time value.

start-field and *end-field* specify the range of DATETIME fields for the operand, as described under [DATETIME Data Type](#) on page D-15. If the range includes fields not in value of the column, literal, function, host variable, or parameter, SQL extends the value to include the new fields by using the same initial values as for an extension using the EXTEND function.

If the range omits fields not in the original value, SQL truncates those fields.

interval-operand

is a column name (possibly qualified by a correlation name), a literal, a function invocation, a host variable, a parameter name, or an expression that evaluates to an INTERVAL value.

start-field, *digits*, and *end-field* specify the range of INTERVAL fields for the operand and the number of digits in the starting field, as described under [INTERVAL Data Type](#) on page I-19.

arithmetic-operator

specifies one of these arithmetic operations:

- ** Exponentiation
- * Multiplication
- / Division
- + Addition
- Subtraction

Considerations—Expressions

- Order of evaluation

The order of evaluation of an expression is:

1. Expressions within parentheses
2. Unary operators
3. Exponentiation
4. Multiplication and division
5. Addition and subtraction

Operators at the same level are evaluated from left to right for all operators except exponentiation. Exponentiation operators at the same level are evaluated from right to left. For example, $X + Y + Z$ is evaluated as $(X + Y) + Z$, whereas $X ** Y ** Z$ is evaluated as $X ** (Y ** Z)$.

- An expression with a numeric operator evaluates to null if any of the operands is null.

Dividing by 0 causes an error.

Exponentiation is allowed only with numeric data types but the operands can be of any numeric type. If the first operand is 0, the second operand must be greater than 0, and the result is 0. If the second operand is 0, the first operand cannot be 0, and the result is 1. If the first operand is negative, the second operand must be an integer.

Exponentiation is subject to rounding error. Results should be considered to be approximate. If your application requires exact values, use the exponentiation function in your host language.

- This paragraphs describe how SQL computes the precision and scale of an arithmetic expression. Precision is the maximum number of digits in the expression. Magnitude is the number of digits to the left of the decimal point. Scale is the number of digits to the right of the decimal point. Precision equals the sum of the magnitude and the scale.

For example, a column declared as NUMERIC (18,5) has a precision of 18, a magnitude of 13, and a scale of 5. This literal has a precision of 9, a magnitude of 5, and a scale of 4:

```
12345.6789
```

SQL computes precision, magnitude, and scale during the evaluation of an expression. When SQL detects an operator in the expression, it applies This rules:

- If the operand is + or –, the resulting scale is the maximum of the scales of the first and second operands. The resulting precision is the maximum of the magnitudes of the first and second operands, plus the scale of the result, plus 1.

If the operator is *, the resulting scale is the sum of the scales of the first and second operands. The resulting precision is the sum of the magnitude of the first operand, the magnitude of the second operand, and the scale of the result.

- If the precision becomes greater than 18, the resulting precision is set to 18. If the expression contained a division operator (/), the resulting scale is the maximum of 0 and (18- (result precision - result scale)). Both operands are truncated to the resulting scale.
- If the operator is /, the resulting precision equals 18 and the resulting scale is the maximum of 0 and (18 - magnitude operand1 - scale operand2).

Consider this expression:

```
(100.00 - ((COL1 * 100.00) / COL2))
```

Suppose that the operands are defined:

```
COL1  LARGEINT  precision=18, scale=0, magnitude=18
COL2  LARGEINT  precision=18, scale=0, magnitude=18
100.00 constant  precision=5,  scale=2, magnitude=3
```

SQL evaluates the expression:

- First, perform the multiplication, $(COL1 * 100.00)$. The resulting scale is $(0 + 2) = 2$. The resulting precision is $(18 + 3 + 2) = 23$. The precision is greater than 18, so it is set to 18. The resulting magnitude is $(18 - 2) = 16$.
- Next, perform the division, $(result / COL2)$. The resulting precision is 18, the resulting scale is $(18 - 16 - 0) = 2$, and the resulting magnitude is $(18 - 2) = 16$.
- Third, perform the subtraction, $(100.00 - result)$. The resulting scale is $MAX(2, 2) = 2$. The resulting precision is $MAX(3, 16) + 1 + 2 = 19$. Result precision is greater than 18, so it is set to 18. A divide was previously done, so the scale becomes $MAX(0, (18 - (19 - 2))) = 1$. The resulting magnitude is $(18 - 17) = 1$.
- SQL automatically converts between floating point numeric types (REAL and DOUBLE PRECISION) and other numeric types. All numeric values in the expression are first converted to binary, with the maximum precision needed for the evaluation. The maximum possible precision for exact numeric data types is 18 digits. The maximum for REAL and DOUBLE PRECISION data types is approximately 16 digits.
- You can use date-time and INTERVAL operands with arithmetic operators only in these combinations:

Operand 1	Operator	Operand 2	Result Type	Notes
Date-time	-	Date-time	INTERVAL	a, b
Date-time	+ or -	INTERVAL	DATETIME	a, c, d
INTERVAL	+	Date-time	DATETIME	a, c, d
INTERVAL	+ or -	INTERVAL	INTERVAL	a, e
INTERVAL	* or /	Numeric	INTERVAL	a, f
Numeric	*	INTERVAL	INTERVAL	a
INTERVAL	/	INTERVAL	Numeric	g

- a. In a date-time or INTERVAL expression, you can specify fields for the result with a range of fields following the expression. For example, this expression gives the result 09-17:

```
( DATE "1988-09-22" - INTERVAL "5" DAY ) MONTH TO DAY
```
- b. If you subtract a date-time value from another date-time value, both values must have the same range of date-time field.
- c. Adding an INTERVAL of MONTHS to a DATE value results in a value of the same day plus the specified number of months. Because different months have different lengths, this is an approximate result.
- d. Date-time and INTERVAL arithmetic that involves MONTH and DAY fields can yield unexpected results, depending on how the fields are used. For example,

this expression (evaluated left to right) generates an SQL error because the calculation must use February 30:

```
DATETIME "1989-01-30" YEAR TO DAY
      + INTERVAL "1" MONTH + INTERVAL "7" DAY
```

In contrast, this expression (which adds the same values as the previous one, but in a different order) generates the value 1989-03-06:

```
DATETIME "1989-01-30" YEAR TO DAY
      + INTERVAL "7" DAY + INTERVAL "1" MONTH
```

Addition or subtraction of a date-time value and an INTERVAL value results in a DATETIME value that must be within the range of fields for the result. SQL adjusts values in adjacent DATETIME fields if necessary.

The result of adding or subtracting an INTERVAL of *n* YEARS to or from a date-time value is a value *n* YEARS after or before the original date-time value. The other fields of the result remain the same.

- e. Truncation occurs if the result of adding or subtracting two INTERVAL values causes a result that does not fit in the receiving field's range of INTERVAL fields. SQL issues a warning if this occurs.
- f. If you multiply or divide an INTERVAL value by a numeric value, SQL converts the INTERVAL value to its smallest subfield and then multiplies or divides it by the numeric value. The range of fields in the result is the minimum range required to contain the final result.
- g. You can only divide an INTERVAL by another INTERVAL if the two INTERVAL values are compatible. You cannot divide a year-month interval by a day-time interval.

Examples—Expressions

- These examples are of arithmetic expressions:

<code>QTY_ON_HAND * AVG (PARTCOST)</code>	Column value multiplied by function applied to column values
<code>QTY_ORDERED * (PRICE - PARTCOST)</code>	Column values combined by operators
<code>PRICE * :INCREASE</code>	Column value multiplied by value in host variable

- In this example and in all these examples, date-time and INTERVAL values are from this table:

Table Definition:

```
CREATE TABLE PROJECTS
  ( PROJECT_NAME      PIC X(10)                NOT NULL ,
    START_DATE        DATETIME YEAR TO MINUTE   NOT NULL ,
    END_DATE          DATETIME YEAR TO MINUTE   NOT NULL ,
    WAIT_TIME         INTERVAL DAY(2)          NOT NULL )
```

Table Data:

PROJECT_NAME	START_DATE	END_DATE	WAIT_TIME
920	1988-02-21:20:30	1989-03-21:20:30	20
134	1970-01-01:00:00	1978-03-21:20:30	30
922	1940-02-21:12:30	1941-03-21:20:30	13
955	1990-10-14:14:30	1991-01-20:12:30	14

- This example adds an INTERVAL value to a DATETIME value. The result is 1942-03-21:20:30.

```
>> SELECT END_DATE + INTERVAL "1" YEAR
+>     FROM PROJECTS WHERE PROJECT_NAME = "922";
```

- This example subtracts an INTERVAL value qualified by MONTH from a DATETIME value. The result is 1990-12-20:12:30. The YEAR value is decremented by 1 because subtracting a month from January 20 causes the date to be in the previous year.

```
>> SELECT END_DATE - INTERVAL "1" MONTH
+>     FROM PROJECTS WHERE PROJECT_NAME = "955";
```

- This example adds an INTERVAL value qualified by DAY to a DATETIME value. SQL handles 1988 as a leap year. The result is 1988-03-12:20:30.

```
>> SELECT START_DATE + WAIT_TIME
+>     FROM PROJECTS WHERE PROJECT_NAME = "920";
```

- This example subtracts an INTERVAL value from a DATETIME value and adjusts the adjacent field. The result is 1940-02-20:21:00.

```
>> SELECT START_DATE - INTERVAL "15:30" HOUR TO MINUTE
+>     FROM PROJECTS WHERE PROJECT_NAME = "922";
```

- This example adds two INTERVAL values:

```
>> INSERT INTO PROJECTS
+>   (PROJECT_NAME, START_DATE, END_DATE, WAIT_TIME)
+>   VALUES ( "945", DATE "1989-10-20" ,
+>           DATE "1990-10-21" ,
+>           INTERVAL "30" DAY + INTERVAL "3" HOUR) ;
```

Because the receiving field has DAY as its range of DATETIME fields, the result of adding 30 days and 3 hours is expressed as 30 days. For the HOUR value to

appear, the WAIT_TIME column must be defined with the range DAY TO HOUR.
The inserted row is:

```
945      1989-10-20:00:00      1990-10-21:00:00      30
```

- This example doubles an INTERVAL value. The result is 5 years, 2 months.

```
INTERVAL "2-7" YEAR TO MONTH * 2
```

- This example divides an INTERVAL value by another. The result is 36.

```
INTERVAL value:
INTERVAL "3" DAY / INTERVAL "2" HOUR
```

- This SQLCI example uses the UNITS clause to convert a date-time field to a numeric field:

```
>> SELECT START_DATE UNITS MONTH FROM PROJECTS;
(EXPR)
-----
      2
      1
      2
     10
     10
--- 5 row(s) selected.
```

- This programmatic example adds 10 years to the start date and stores the result in a DATETIME variable. The statement uses the UNITS clause to convert a numeric 10 to a value of type INTERVAL YEAR.

```
EXEC SQL SELECT STARTDATE + 10 UNITS YEAR
        INTO :NEWDATE TYPE AS DATETIME YEAR TO MINUTE
        FROM   PROJECTS WHERE PROJECTNAME = '920';
```

EXTEND Function

EXTEND is a function that adjusts the range of fields for a date-time value to a specified range or to the default range of DATETIME fields.

It returns a value of type DATETIME.

```
EXTEND ( date-time-expression
        [ , [start-date-time TO ]end-date-time] )
```

date-time-expression

is an expression that evaluates to a value of type DATETIME, DATE, TIME, or TIMESTAMP.

`[start-date-time TO] end-date-time`

is a range of DATETIME fields (for example, YEAR TO DAY). If the range is not specified, the system uses YEAR TO FRACTION(6).

Considerations—EXTEND

- Any fields in *date-time-expression* that are not in the specified range are truncated.
- If the range contains fields to the left of the fields in *date-time-expression*, the additional fields receive values based on the current date or time. If the result is not a valid DATETIME value, an SQL error is generated.
- If the range contains fields to the right of the fields in *date-time-expression*, the additional fields are initialized:

Field	Initial Value
MONTH	01
DAY	01
HOUR	00
MINUTE	00
SECOND	00
FRACTION	000000

Examples—EXTEND

- In this example, the fields DAY, HOUR, MINUTE, SECOND, and FRACTION to the right of MONTH are initialized to 01 (for DAY), 00 (for HOUR, MINUTE, and SECOND) and 000000 (for FRACTION):

```
EXTEND (DATETIME "1994-11" YEAR TO MONTH , YEAR TO FRACTION)
```

The function returns the value:

```
1994-11-01:00:00:00.000000
```

- In this example, the field YEAR to the left of MONTH is initialized to the current year. The HOUR and MINUTE fields to the right of MONTH are initialized to 0:

```
EXTEND ( DATETIME "11-24" MONTH TO DAY , YEAR TO MINUTE )
```

In 1994, the function returns the value:

```
1994-11-24:00:00
```

EXTENT File Attribute

EXTENT is a Guardian file attribute that sets the size of the extents (units of contiguous disk space) that will be allocated for a file or a partition of a file. EXTENT applies to key-sequenced, relative, and entry-sequenced tables and to indexes.

EXTENT is set when a file or partition is created. Each partition of a partitioned file has its own EXTENT attribute that can differ from the EXTENT attribute for other partitions of the file. You can specify a single extent size for each extent in the file or partition, or you can specify one size for the primary (first) extent and another size for the secondary extents.

```
EXTENT { ext-size
        { ( pri-ext-size [, sec-ext-size ] ) }

ext-size, pri-ext-size, or sec-ext-size is:

integer [ PAGE[S]      ]
        [ BYTE[S]      ]
        [ REC[S]       ]
        [ MEGABYTE[S]  ]
        [ GIGABYTE[S]  ]
```

The default is 16 pages for the primary extent and 64 pages for each secondary extent.

The default unit type is PAGE.

integer

is an integer that specifies the number of pages, bytes, recs, or megabytes in the extent. The ranges allowed for Format 1 partitions are:

PAGES	0 to 65,535
BYTES	0 to 134,215,680
RECS	0 to 134,215,680
MEGABYTES	0 to 134

The ranges allowed for Format 2 partitions are:

PAGES	0 to 512,000,000
BYTES	0 to 1,048,576,000,000
RECS	0 to 1,048,576,000,000
MEGABYTES	0 to 1,048,576
GIGABYTES	0 to 1,048

A PAGE consists of 2048 bytes.

A MEGABYTE consists of 1,000,000 bytes. A GIGABYTE consists of 1,000,000,000 bytes.

Considerations—EXTENT

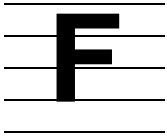
- A file's extent size must be at least as large as its block size and must be a multiple of the block size and a multiple of page size (2048 bytes). If you specify extent sizes that do not meet these conditions, SQL uses the next block size or the next full page size. For example, 0 PAGE rounds up to 1 PAGE.

A file (or a partition of a partitioned file) must fit on a disk, so the size of the primary extent plus the total size of all secondary extents must not exceed the disk size.

- A primary extent should be large enough to hold the file at the initial load, and secondary extents should be large enough to accommodate growth. The faster the growth, the larger the secondary extents should be.

To ensure adequate space for your file, choose extent sizes and a MAXEXTENTS value large enough to accommodate the amount of data you expect to store in the file.

Using large extents can improve performance by reducing the number of seeks. The disadvantage of large extents is that an entire extent is allocated simultaneously, leaving allocated but unused space on the disk while the extent contains only a small amount of data. You can maximize the use of disk space by specifying smaller extent sizes if performance is not an issue.



FC Command

FC is an SQLCI command with which you can retrieve, edit, and reexecute a command in the history buffer. For more information, see [HISTORY Command](#) on page H-5.

```
FC [ text      ] [ ; ]  
   [ number    ]  
   [ -number   ]
```

text

specifies the most recent version of a command in the history buffer. The command must begin with the text that you specify; you need only the characters necessary to identify the command. The text can be in uppercase or lowercase characters.

number

is a positive integer that refers to the ordinal number of a command in the history buffer.

-number

is a negative number that indicates the position of a command in the history buffer relative to the current command entered.

Considerations—FC

- To retrieve the last statement or command entered, you can enter FC without specifying text or a number.
- The IN file must be a terminal or a process. If you put more than one statement or command on an input line, FC must be last.
- The command you specify with FC is displayed one line at a time. As each line appears, you can modify it by entering these editing commands:

D	Deletes the character immediately above the D. Repeat to delete more characters.
I <i>characters</i>	Inserts characters in front of the existing character that is immediately above the I.
R <i>characters</i>	Replaces existing characters with the specified characters one-by-one, beginning with the character immediately above the R.

characters (Must begin with a nonblank character) replaces existing characters with the specified characters one-for-one, beginning with the character immediately above the first character specified.

To terminate a command and to specify more than one editing command on a line, separate the editing commands with a double slash (/).

When you have no further changes to make, press the Return key. The command prints again to allow you to edit it again. To stop editing, press the Return key without entering any editing commands.

After you modify the last line and accept it by pressing the Return key, the revised command executes.

- To abnormally end the FC command and leave the original command unchanged, press the Break key or Ctrl-Y, or enter a double slash (/) in columns 1 and 2, followed immediately by a Return key.

Examples—FC

- To correct the last statement or command entered, use FC without text or a number:

```
>> EXEUCTE SELSUPP USING ?ST = "CALIFORNIA";
    ^
*** ERROR from SQLCI [-10021] Syntax error.
>> FC
>> EXEUCTE SELSUPP USING ?ST = "CALIFORNIA";
..    CU
>> EXECUTE SELSUPP USING ?ST = "CALIFORNIA";
..
```

Press the Return key to execute the corrected version.

- Suppose that you want to display information about a specific supplier. You enter a query to select supplier number 4, and learn that the supplier number does not exist. You then decide to list all suppliers, but want only five rows displayed at a time.

```
>> SELECT SUPPNAME, CITY, STATE FROM INVENT.SUPPLIER
+> WHERE SUPPNUM = 4;
-- 0 row(s) selected.
>> SET LIST_COUNT 5;
```

To execute the query again but save typing effort, you can use the FC command, specifying the relative position of the SELECT command in relation to the current command, FC:

```
>> FC -2
>> SELECT SUPPNAME, CITY, STATE FROM INVENT.SUPPLIER
.
+> WHERE SUPPNUM = 4;
```



```
. DDDDDDDDDDDDDDDDDDD
+> ;
```

The first five rows of the table SUPPLIER are listed.

- Suppose that you have executed several commands since you entered the SELECT command. You can either use the HISTORY command to determine the SELECT command number, or you can enter:

```
>> FC SEL
```

If the history buffer contains several SELECT commands, you must be more specific; for example, FC SELECT SUPP.

- To edit and reexecute the command numbered 14 in the history buffer, enter:

```
>> FC 14
```

FETCH Statement

FETCH is a DML and dynamic SQL statement that returns a value for each column in the next row of the result table defined by the cursor, leaving the cursor positioned at that row. FETCH can be used only in host programs.

```
FETCH { cursor } { INTO :var [ , :var ] ... }
      { :cursor-var } { USING DESCRIPTOR :sqlda-desc }
```

cursor

is the name of an open cursor.

:*cursor-var*

(dynamic SQL only) is a host variable of SQL type CHAR or VARCHAR that stores the name of an open cursor.

INTO :*var* [, :*var*] ...

identifies one or more host variables to receive values. FETCH returns one SELECT item per host variable. The data type of each variable must be compatible with the data type of the corresponding SELECT column.

Use this option in dynamic SQL if you know the number and data types of the returned columns.

USING DESCRIPTOR :*sqlda-desc*

(dynamic SQL only) is a host variable containing an SQLDA descriptor that describes a list of memory locations (not always declared host variables) into which corresponding SELECT columns are copied.

Use this option in dynamic SQL if you have no previous knowledge of the returned columns and use DESCRIBE to retrieve their descriptions.

Considerations—FETCH

- FETCH requires read access to any tables or views associated with the cursor. Updating fetched rows requires write access to the table or view.
- Successive executions of FETCH retrieve successive rows in the result table. To control the order in which the rows appear, include an ORDER BY clause in the SELECT portion of the DECLARE CURSOR statement that defines the cursor.
- If the number of host variables is different from the number of columns in the result table, FETCH issues a warning and returns the number of values in the shorter list (column list or host variable list).

If the column list is shorter than the host variable list, the values in the extra host variables are indeterminate.

- Locking occurs when the FETCH executes (or, if the SELECT requires a sort, when you open the cursor), but the SELECT statement associated with the cursor specifies whether the access option that controls locking is BROWSE, STABLE, or REPEATABLE.

A FETCH for a cursor on an audited table that uses STABLE or REPEATABLE access must execute in the same TMF transaction that opened the cursor.

- FETCH returns an integer status code to SQLCODE:

0	The FETCH was successful
100	The end of the table was encountered
> 0	A warning was issued
< 0	An error occurred

Avoid using SQLCODE 100 as an end-of-file indicator. SQL resets SQLCODE to 0 when you close the associated cursor. Instead, define your own end-of-file flag.

Examples—FETCH

- In this example, suppose that you have a cursor that returns information from the PARTS table. The host variables are declared in a Declare Section, and the cursor declaration lists the columns to be retrieved. The FETCH statement lists a corresponding host variable to receive the values returned for each column. (The example uses the SQL statement terminator for COBOL programs.)

Variable declarations:

```

...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
Declare host variables:
    01 HVAR1
    02 HVAR2
    03 HVAR3
EXEC SQL END DECLARE SECTION END-EXEC.

```

```

...
Main code:
EXEC SQL DECLARE CURSOR1 CURSOR FOR
    SELECT COL1,
           COL2,
           COL3
    FROM   =PARTS
    WHERE  COL1 >= :HOSTVAR1
    ORDER BY COL1
    BROWSE ACCESS
END-EXEC.

```

```

...
EXEC SQL OPEN CURSOR1 END-EXEC.
EXEC SQL FETCH CURSOR1
           INTO   :HVAR1,
                  :HVAR2,
                  :HVAR3
END-EXEC.
EXEC SQL CLOSE CURSOR1 END-EXEC.

```

- These steps demonstrate a dynamic SQL FETCH. The code uses FETCH with USING DESCRIPTOR to return information on SELECT columns of which there is no previous knowledge. (The example uses the SQL statement terminator for C, Pascal, and TAL programs.)

1. Declare an SQLDA to hold input parameters and name the area SDAI; NAMESINPUT is the names buffer. The values 5 and 39 are arbitrary values chosen for the size of the SQLDA and the names buffer. Your program can use different values or allocate memory dynamically.

```
EXEC SQL INCLUDE SQLDA (SDAI, 5, NAMESINPUT, 39);
```

2. Declare an SQLDA to hold output variables (SELECT columns) and name it SDAO; NAMESOUTPUT is the names buffer:

```
EXEC SQL INCLUDE SQLDA (SDAO, 5, NAMESOUTPUT, 39);
```

3. Read an SQL statement input from the terminal and store it in variable H1.
4. Prepare the input as S1:

```
EXEC SQL PREPARE S1 FROM :H1;
```

5. Fill in the SQLDA and names buffer with the descriptions of the parameter values (input parameters) in the SQL statement:

```
EXEC SQL DESCRIBE INPUT S1 INTO :SDAI
      NAMES INTO :NAMESINPUT;
```

6. Fill in the SQLDA and names buffer with the descriptions of the SELECT columns (output variables) in the SQL statement:

```
EXEC SQL DESCRIBE S1 INTO :SDAO NAMES INTO :NAMESOUTPUT;
```

7. Declare a cursor, C1, for the statement S1. Open the cursor using the parameter values that were input and stored in :SDAI:

```
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1 USING DESCRIPTOR :SDAI;
```

8. Retrieve the column values stored in :SDAO:

```
EXEC SQL FETCH C1 USING DESCRIPTOR :SDAO;
```

File Attributes

File attributes describe the physical characteristics of a file or an SQL object, such as a table or an index, that is stored in a file. The values you select for a file's attributes can affect the storage and security for the object and the performance of applications that use the object.

File attributes are set when a file is created. If you do not specify attribute values in the statement that creates an SQL object (such as CREATE TABLE or CREATE INDEX), SQL uses default values. Many attributes can be changed later (with statements such as ALTER TABLE or ALTER INDEX), some attributes remain in effect for the life of the object, and a few can change as a side effect of a command or a change to some other attribute.

This table summarizes the file attributes important for SQL objects. Because SQL objects reside in Guardian files, all the attributes listed are attributes of Guardian files. For more information, see the descriptions of specific attributes.

File Attributes of SQL Objects

ALLOCATE	Controls amount of disk space allocated. Default is to allocate space as needed.
AUDIT	Controls TMF auditing. Default is AUDIT.
AUDITCOMPRESS	Controls whether unchanged columns are included in audit records. Default is to include only changed columns.
BLOCKSIZE	Sets size of data blocks. Default is 4096.
BUFFERED	Turns buffering on or off. Default is on.
CLEARONPURGE	Controls disk erasure when file is dropped. Default is no erasure.
DCOMPRESS	Controls key compression in data blocks. Default is no compression.
DSLACK	Sets percent of slack in data blocks. Default is value of the SLACK attribute.
EXTENT	Sets extent sizes. Default is 16 pages for the first extent, 64 for others.
FORMAT	Controls whether a partition is Format 1 or Format 2.
ICOMPRESS	Controls key compression in index blocks. Default is no compression.
ISLACK	Sets percent of slack in index blocks. Default is value of the SLACK attribute.
LOCKLENGTH	Sets number of bytes in key to use for generic locks. Default is entire key.
MAXEXTENTS	Sets maximum extents. Default is 160.
NOPURGEUNTIL	Sets date after which drop is allowed. Default allows immediate drop.
OWNER	Specifies owner. Default is creator.
PROGID	Determines the PAID of a process started from the file. Default is NO PROGID.
RECLENGTH	Sets bytes reserved for a relative-file row. Default is total column lengths.
RESETBROKEN	Resets BROKEN flag. No default.
SECURE	Sets Guardian security string. Default is creator's default security string.
SERIALWRITES	Specifies serial or parallel writes. Default is SERIALWRITES.

SLACK	Sets percent of slack in blocks if not specified by DSLACK or ISLACK. Default is 15 percent.
TABLECODE	Sets tablecode. Default is 0.
VERIFIEDWRITES	Controls verification of writes to disk. Default is no verification.

File Organizations

SQL DDL statements create and modify tables and indexes and the physical Guardian files that hold tables and indexes. To select parameters for your DDL statements, you must be familiar with the three physical file organizations available for SQL tables: key-sequenced, entry-sequenced, and relative.

- In key-sequenced files, records are stored in sequence by primary key or clustering key. The key can be supplied by the user, generated by the system, or built from values supplied by the user and a value generated by the system. You cannot update columns in a primary or clustering key.

You can insert, update, or delete data in rows, shorten or lengthen values in varying-length character columns, and alter table definitions to add columns. You can also add, move, or drop partitions.

Tables are often stored in key-sequenced files, and indexes are always stored in key-sequenced files.

- In entry-sequenced files, each new record is added to the logical end of the file. The primary key is a system-generated record address. You can add or update rows, but you cannot delete them. You cannot shorten or lengthen values in varying-length character columns and you cannot alter table definitions to add columns. You can add or move partitions, but you cannot drop partitions.
- In relative files, records are stored at relative record locations specified by either the user or the file system. The primary key is the relative record number. You can insert, update, or delete rows, and you can shorten or lengthen values of varying-length character columns. You can alter a table definition to add columns if the original record length defined for the table is large enough to include the added columns.

Guardian files that do not contain SQL objects but that have key-sequenced, entry-sequenced, or relative file organization are also called *Enscribe files* or *structured files*.

A fourth type of Guardian file—unstructured—is also used on NonStop systems, but never for SQL tables or indexes. SQL programs in Guardian files are stored in unstructured files. Edit files—text files that can be read by the EDIT or TEDIT text editors and by many other Guardian utilities—are unstructured files with file code 101. (OSS users can convert files created with the vi text editor to EDIT files with the CTOEDIT command described in the *Open System Services Programmer's Guide*.)

For information about choosing the most effective file organizations for your tables, see the *SQL/MP Installation and Management Guide*.

FILEINFO Command

FILEINFO is an SQLCI utility that displays information about the versions and physical characteristics of tables, indexes, views, collations, Enscribe files, and OSS files.

FILEINFO is similar to FUP INFO, although its displays are slightly different.

```
FILEINFO qualified-fileset-list

[[,]fileinfo-option] ... ;

fileinfo-option is:}

{ USER [ group.member ] }
{      [ group-ID,member-ID ] }
{      [ "user-name" ] }
{      [ user-ID ] }
{ [ BRIEF | DETAIL ] }
{ EXTENTS }
{ STAT[ISTICS] [ , PARTONLY ] }
{ SHADOWS }
```

qualified-fileset-list

is a qualified fileset list that specifies files for which to display information. For information, see [Qualified Fileset List](#) on page Q-1.

You cannot include a pathname in a qualified fileset list; use Guardian-format ZYQ names to specify OSS files.

```
USER [ group.member ] }
      [ group-ID,member-ID ] }
      [ "user-name" ] }
      [ user-ID ] }
```

restricts the files for which information is displayed to those from *qualified-fileset-list* that are owned by the specified user.

You can specify user in one of several ways:

group.member is a valid user name.

group-id, member-ID specifies the user ID as a pair of numbers, each in the range from 1 to 255.

"*user-name*" is a valid Guardian user name (in *group.member* format) or Safeguard alias name, enclosed in double quotes. To add a Safeguard alias, use the Safecom utility.

user-ID is a user's numeric ID such as that displayed by the OSS ls -n command. *user-ID* can be in the range from 1 to 65,535. For a Guardian *user-ID*, this number equates to $(256 * \text{group-id} + \text{member-ID})$.

If you specify a user, FILEINFO displays information for files owned by that user. If you specify only the keyword USER, FILEINFO displays information only for files that you own. If you omit the USER option, FILEINFO displays information about all files in *qualified-fileset-list*.

BRIEF | DETAIL

specifies whether to display brief or detailed information about each file. BRIEF, the default, displays only a single line of information.

EXTENTS

displays information on the allocation of extents for each file. Extent information appears for tables, indexes, collations, and Enscribe files, but not for views or OSS files.

STAT[ISTICS] [, PARTONLY]

provides all the DETAIL information and statistical data on blocks and records for each file. Statistics information appears only for tables, indexes, collations, and Enscribe structured files; it does not appear for shadow labels or OSS files.

PARTONLY limits the information to the partitions you specify explicitly in *qualified-fileset-list*. For example, if you specify only a secondary partition of a table, no statistical information about the primary partition or any other secondary partition appears. PARTONLY affects only the output and has no effect on how SQL reads the input file.

If you omit PARTONLY and specify the primary partition of an Enscribe file or specify any partition of an SQL object, FILEINFO supplies information about all partitions. For Enscribe files, PARTONLY is implied if you specify a secondary partition, because secondary partitions do not contain information about other partitions.

SHADOWS

specifies that you want to display information about shadow labels, temporary file labels for objects that have been dropped.

You can specify the DETAIL, EXTENTS, and STATISTICS display together in the same command.

Considerations—FILEINFO

- FILEINFO without the STATISTICS option has no authorization requirements. With the STATISTICS option, FILEINFO requires read authority for the files for which information is displayed.
- FILEINFO returns error -9132 if it encounters an SQL object (but not an SQL program or catalog) that has a version greater than the version of the SQL/MP software on the node from which you issued the FILEINFO command. Use a later version of FILEINFO (that is, a later version of NonStop SQL/MP) to display information about such an object.

FILEINFO returns error -9133 if it encounters an SQL object (but not an SQL program or catalog) that has a version greater than the version of the SQL/MP software on the node on which the object resides. No version of FILEINFO can access such an object, which might exist because of a fallback situation.

BRIEF Display for SQL Objects and Guardian Files

The BRIEF display for SQL objects and Guardian files includes three types of lines:

- A heading line, such as,

```
CODE   EOF   LAST MODIF  OWNER   RWP  TYPE   REC   BLOCK
```
- A line that names the node, volume, and subvolume for the files that follow in the report, such as,

```
[ \node. ]$volume.subvol
```
- An information line that provides information about a specific file.

The information line begins with the unqualified name and the open state of the table, index, collation, or file. The remainder of the line includes information that corresponds to each of the fields in the heading line, as described in [Table F-1](#).

Table F-1. FILEINFO BRIEF Display Information Line (page 1 of 3)

Open State	The open state of the file is described by one or more of these codes:	
B	File is marked broken because of a detected inconsistency in the file structure. (Not used for views.)	
C	Either the data or the definition of the object is corrupt. (DETAIL display shows which one is corrupt.)	
O	File is open. (Not used for views.)	
R	Recovery is needed. See the <i>TMF Operations and Recovery Guide</i> .	

Table F-1. FILEINFO BRIEF Display Information Line (page 2 of 3)

Open State	<p>The open state of the file is described by one or more of these codes:</p> <p>D An ALTER TABLE or ALTER INDEX operation using the WITH SHARED ACCESS option did not complete successfully. To recover, use the RECOVER INCOMPLETE SQLDDL OPERATION option for the ALTER TABLE or ALTER INDEX statement, followed by a FUP RELOAD operation. For more information, see the ALTER TABLE or ALTER INDEX command or the WITH SHARED ACCESS option.</p> <p>F An ALTER TABLE or ALTER INDEX operation using the WITH SHARED ACCESS option left unreclaimed free space on disk. To recover the free space, issue a FUP RELOAD command. For more information about FUP RELOAD, see the <i>File Utility Program (FUP) Reference Manual</i>.</p> <p>S File is a shadow label, a temporary file label for an object that has been dropped which exists until the transaction is committed, aborted, or rolled back, or until a TMF file recovery is performed. (Shadow labels are not listed if the fileset is qualified by a catalog name.)</p> <p>? File is in the crash-open or crash-label state. A file is in the crash-label state if a file label operation was taking place at the time of a total system crash or when the disk on which the file is located becomes unavailable. A file is in the crash-open state if it was open in the same circumstances. (Only the crash-label state applies to views.)</p> <p>blank File is not open, crashed, or broken.</p>
CODE	<p>The file code, or "OSS" for an OSS file. The default Guardian file code of 0 is not displayed.</p> <p>Guardian file codes 100 through 999 refer to specific types of files and are reserved by the HP NonStop operating system. For the meanings of reserved file codes, see the description of the FUP INFO command in the <i>File Utility Program (FUP) Reference Manual</i>.</p> <p>Letters that follow the file code have specific meanings:</p> <p>A File is audited by the TMF subsystem.</p> <p>L File is licensed by the super ID.</p> <p>P PROGID security attribute of the file is on.</p> <p>+ This partition is Format 2. If blank, this partition is Format 1.</p>
EOF	The number of bytes in the file. (Not used for views.)
LAST MODIF	The date and time when the file was last modified. If the last modification date is the current date, only the time of day is given. (Not used for views.)
OWNER	The user ID of the file's owner.
RWEP	The security string for the file.

Table F-1. FILEINFO BRIEF Display Information Line (page 3 of 3)

Open State	The open state of the file is described by one or more of these codes:	
TYPE	A six-character code that identifies the file type and SQL object type in this form:	
	Character positions -> 1 2 3 4 5 6	
	Codes that appear -> X P E A In	
	K P Pg	
	R S Ta	
	Vi	
	Character Position	Code Meaning
	1	X Secondary Partition
	2	P Partitioned Table, protection view or file
	3	E Entry-sequenced file structure
		K Key-sequenced file structure
		R Relative file structure
		Blank View or unstructured file
	4	A Enscribe file with alternate key
		P Protection view
		S Shorthand view
		Blank Tables, indexes, collations, and files without alternate keys
	5-6	In Index
		Pg SQL object program file
		Ta Table
		Vi View
		Blank Collations and Enscribe files that are not SQL object program files. (To determine which, use the DETAIL option.)
REC	The record length in bytes. For tables and indexes, REC is the maximum record length; for Guardian files, REC is the logical record length. (Not displayed for views or unstructured files.)	
BLOCK	The number of bytes in each block of a file. (Not displayed for views or unstructured files.)	

DETAIL Display for Objects (Except Views) and Guardian Files

This listing shows all the detail that can appear if you specify the DETAIL option when you request FILEINFO for a table, index, collation, or Guardian file. The information that actually appears in your listing depends on the file organization of the item you inquire about and on whether the item is a table, index, collation, or file.

The KEY descriptors, INDEX and ALTKEY, and PART information do not appear for shadow labels. The SMF information appears only for files managed by SMF.

The name of the object or file and the date when the listing is produced appear in the first line. The numbers in the left most column do not appear in the listing. These numbers relate sections of the listing to the notes and explanations that follow.

- | | <i>file-name</i> | <i>date-and-time</i> |
|-----|--|----------------------|
| 1. | <i>object-type</i> | |
| 2. | CATALOG <i>catalog-name</i> | |
| 3. | VERSION <i>version-number</i> | |
| 4. | PROGRAM CATALOG VERSION <i>version-number</i> | |
| 5. | PROGRAM FORMAT VERSION <i>version-number</i> | |
| 6. | BASE TABLE <i>base-table-name</i> | |
| 7. | TYPE <i>organization-type</i> | |
| 8. | FORMAT <i>format-type</i> | |
| 9. | CODE <i>file-code</i> | |
| 10. | EXT (<i>pri-num</i> PAGES, <i>sec-num</i> PAGES, MAXEXTENTS <i>max-extents</i>) | |
| 11. | REC <i>record-length</i>
PACKED REC <i>packed-record-length</i>
RECLength <i>max-record-length</i>
BLOCK <i>block-length</i> | |
| 12. | IBLOCK <i>block-length</i>
KEY (<i>key-descriptor</i>)
SYSKEY
LOCKLENGTH <i>lock-length</i>
DCOMPRESS, ICOMPRESS | |
| 13. | For each index or alternate key of the object or file: | |
| 14. | {INDEX} (<i>key-spec</i> , FILE <i>alt-fnum</i> , <i>file-name</i> ,
{ALTKEY} <i>key-descriptor</i>
{UNIQUE NO UNIQUE},
{UPDATE NO UPDATE}, NULL <i>null-value</i>) | |
| 15. | For each partition of the object or file: | |

16. PART (*part-num*, *\$volume*,
pri-ext PAGES, *sec-ext* PAGES, MAXEXTENTS
max-ext, FORMAT *format-type*, *firstkey-value*)
17. ODDUNSTR
REFRESH
AUDIT
BUFFERSIZE
BUFFERED
AUDITCOMPRESS
VERIFIEDWRITES
SERIALWRITES
INCOMPLETE SQLDDL OPERATION
UNRECLAIMED FREESPACE
18. OWNER *group-id*, *owner-id*
SECURITY (RWE_P):*rwe_P*, PROGID,
CLEARONPURGE, LICENSE
NOPURGEUNTIL:*expire-time*
19. SECONDARY PARTITION
20. MODIF: *modif*, *open-state*
CREATION DATE: *create-time*
REDEFINITION DATE: *redefinition-time*
LAST OPEN: *last-open-time*
21. EOF *eof* (*percent-used* % USED)
22. EXTENTS ALLOCATED: *num-ext*
23. INDEX LEVELS: *num-index-levels*
24. PARTITION ARRAY {EXTENDED | STANDARD | FORMAT2ENABLED}
25. Files managed by SMF,

LOGICAL NAME: *logical-file-name* for physical files
PHYSICAL NAME: *physical-file-name* for logical files

This list describes the items in the DETAIL display:

1. The *object-type* value describes what type of object the file is:
 - SQL table or view
 - catalog table
 - collation
 - index
 - catalog index
 - Enscribe file
 - Enscribe file containing an SQL object program
 - OSS file
 - OSS file containing an SQL object program

- INVALID indicates that an SQL program is not valid and might need to be SQL compiled.
 - SHADOW LABEL indicates that the file is a shadow label.
 - SQL CHARACTER PROCESSING RULES OBJECT indicates that the object is a collation.
2. CATALOG identifies the catalog in which the object is defined.
 3. VERSION indicates a NonStop SQL/MP version number. This information is supplied only for SQL objects.
 4. PROGRAM CATALOG VERSION (PCV) indicates the oldest version catalog in which the SQL program can be registered. The PCV of a program depends on the program's use of SQL/MP features that require information to be recorded in the catalog. This information is supplied only for SQL programs.
 5. PROGRAM FORMAT VERSION (PFV) indicates the oldest version of SQL/MP software that can execute the SQL program. The PFV of a program is the SQL/MP software version of the SQL compiler that compiled the program. This information is supplied only for SQL programs.
 6. BASE TABLE is the underlying table if the file is an index.
 7. TYPE indicates the file organization:
 - K Key-sequenced
 - E Entry-sequenced
 - R Relative
 - U Unstructured
 8. FORMAT indicates the partition format, 1 or 2.
 9. CODE is the file code. For more information, see the discussion of file code in the preceding description of the BRIEF display.
 10. EXT lists the sizes of the primary and secondary extents and the maximum number of extents that can be allocated.
 11. Items in this section do not appear for unstructured files:
 - REC indicates the maximum exploded record length for objects. (A record in “exploded format” is expanded to its maximum length; varying-length character fields are padded with blanks to their full maximum size and filler is generated where necessary to align numeric fields that require word alignment.)
 - PACKED REC indicates the maximum packed record length for objects. (Records on disk are stored in packed format. A record in packed format has no fillers; varying length character fields are stored as their exact size plus the length field and numeric items are not necessarily word aligned.)
 - RECLENGTH indicates the maximum row length for relative tables. For more information, see [RECLENGTH File Attribute](#) on page R-1.
 - BLOCK indicates the number of bytes in a block.

12. This section describes the primary key of a key-sequenced file or other structured file types:

- IBLOCK is the length of an index block of an Enscribe file.
- KEY key-descriptor is one or more sets of these items; the number of sets depends on the number of columns in the key:

```
COLUMN col-num, OFFSET key-offset,
      LENGTH key-length, { ASC }
                        { DESC }
```

- COLUMN number indicates the position of the key column in the row. If the row contains a system-defined primary key, the primary key is column 0; otherwise, the first column defined for the table is column 0.
- OFFSET indicates the zero-relative byte address of the key column in the record.
- LENGTH indicates the length of the key column in bytes.
- ASC is ascending order and DESC is descending order.
- SYSKEY indicates a system-defined primary key.
- LOCKLENGTH is the number of bytes of the primary or clustering key used for locking (including the SYSKEY if it exists).
- DCOMPRESS indicates keys in data blocks of the file are compressed.
- ICOMPRESS indicates keys in index blocks are compressed.

13. This section describes indexes of a table or describes alternate key files for an Enscribe file:

- The *key-spec* value is the key specifier stored in every row of the index. For more information, see [CREATE INDEX Statement](#) on page C-142.
- FILE indicates the number of an Enscribe alternate key file.
- The *key-descriptor* value of the index or alternate key file is displayed as described for the KEY item in note 11.
- UNIQUE or NO UNIQUE indicates whether the indexed column or columns can have the same value or set of values in two or more rows.
- UPDATE or NO UPDATE indicates whether key specifiers of Enscribe files are automatically updated.
- NULL indicates the null value set for an Enscribe file key.

14. If the object or file is partitioned, the partitions are described in this section. The partition number and volume name of each partition are followed by the number of primary and secondary extents and the maximum extent size allowed. The partition's format, 1 or 2, is shown. For a key-sequenced file, the FIRST KEY value is given.

If a file uses multibyte characters, the FIRST KEY value might contain characters unsupported by your terminal, causing unpredictable results in the screen display.

15. This section describes file attributes and flags:

ODDUNSTR	Enscribe odd unstructured file.
REFRESH	File label is updated when file control block changes.
AUDIT	File is audited.
BUFFERSIZE	Default internal transfer size of unstructured file.
BUFFERED	Writes to file are buffered.
AUDITCOMPRESS	Compressed audit-checkpoint messages are generated for files.
VERIFIEDWRITES	Writes to file are verified.
SERIALWRITES	Serial mirror writes are done.
INCOMPLETE SQLDDL OPERATION	An ALTER TABLE or ALTER INDEX operation with the WITH SHARED ACCESS option is in process or has failed. If the operation has completed and the flag remains, recovery is needed. To recover the file, use the PARTONLY RECOVER INCOMPLETE SQLDDL OPERATION option for the ALTER TABLE or ALTER INDEX statement, then do a FUP RELOAD operation. For more information about FUP RELOAD, see the <i>File Utility Program (FUP) Reference Manual</i> .
UNRECLAIMED FREESPACE	A DDL operation with the WITH SHARED ACCESS option left unreclaimed free space on disk. To recover the free space, issue a FUP RELOAD command. For more information about FUP RELOAD, see the <i>File Utility Program (FUP) Reference Manual</i> .

16. OWNER is the user ID of the file's owner. This section also displays the security string of the file, indicates whether the PROGID and CLEARONPURGE attributes are set, and indicates the time after which you can purge the file. LICENSE indicates a licensed file; asterisks (*) indicate a file protected by the Safeguard security product.

17. SECONDARY PARTITION indicates the file is a secondary partition.

18. This section lists dates and times of file activity. MODIF indicates when the file was last modified and one of more of these open states, if applicable:

BROKEN	File is marked as broken.
CORRUPT	Error occurred during a utility operation such as restoring or duplicating.
DEFINITION INVALID	Data or definition of the object is invalid.
LABEL	File is in a crash-label state.
QUESTIONABLE	
OPEN	File is open.
QUESTIONABLE	File is in a crash-open state.
RECOVERY NEEDED	File cannot be opened; volume recovery is needed (TMF2 files only).
REDO NEEDED	File cannot be opened; TMF redo file recovery is needed.
UNDO NEEDED	File cannot be opened; TMF undo file recovery is needed.
CREATION DATE	Indicates the date the table or index was created.
REDEFINITION TIME	Indicates when a change to the table or index caused an SQL program to be recompiled.
LAST OPEN	Indicates the last time the table or index was opened.

For more information, see the discussion of the Open State field under [BRIEF Display for SQL Objects and Guardian Files](#) on page F-11.

19. For unstructured files, EOF is the *end-of-file* pointer containing the relative byte address of the byte after the last significant data byte.

For structured files, EOF is the relative byte address of the first byte of the next available block.

percent-used is the percent of available file space currently used based on available space if all extents are allocated.

20. EXTENTS ALLOCATED is the number of extents currently allocated. This information is supplied only for Guardian files.
21. For key-sequenced files, INDEX LEVELS indicates the number of index levels used for index blocks.

22. For tables and indexes, PARTITION ARRAY indicates the type of array used for the base table and any associated indexes. Possible values are:

EXTENDED	Longer partition array available on versions 320 and later of NonStop SQL/MP.
STANDARD	Partition array used by default by NonStop SQL/MP.
FORMAT2ENABLED	Partition array that allows partitions to be any combination of Format 1 or Format 2 partitions. FORMAT2ENABLED implies that the partition array is also EXTENDED.

23. For files managed by SMF, this section indicates the corresponding file name. Possible values are:

- LOGICAL NAME, *virtual-volume-name.subvolume.file-name* displayed when you execute FILEINFO, DETAIL for a physical file
- PHYSICAL NAME, *physical-volume-name.subvolume.file-name* displayed when you execute FILEINFO, DETAIL for a logical file

DETAIL Display for Views

This figure shows all the information that can appear if you specify the DETAIL option when you request FILEINFO for a view.

```

file-name                                date-and-time
object-type
CATALOG catalog-name
VERSION version-number
BASE TABLE base-table-name
PART ( [\system.] $volume )
...
OWNER group-id,user-id
SECURITY (RWP): rwp
[ LABEL QUESTIONABLE ]
[ DEFINITION INVALID ]
[ REDO NEEDED, UNDO NEEDED ]
CREATION DATE: creation-date
NOPURGEUNTIL: expire-time
REDEFINITION DATE: redefinition-date
INSERTABLE

```

Fields with the same names as previously described fields for tables, indexes, collations, and Guardian files contain similar information.

For protection views, BASE TABLE indicates the name of the underlying table. PART and REDEFINITION DATE appear for protection views only. If the partition is on the current system, the system name does not appear. INSERTABLE appears for views for which you can insert and update values.

LABEL QUESTIONABLE, DEFINITION INVALID, REDO NEEDED, and UNDO NEEDED are open states. See the description of the MODIF field of the DETAIL format for tables, indexes, collations, and Guardian files in the preceding list (item 17).

BRIEF and DETAIL Display for OSS Files

For OSS files, both the BRIEF and DETAIL displays include the ZYQ name, the pathname (wrapped over multiple lines, if necessary), the code (OSS), the EOF location, the last modification date, the owner, and the OSS security information. FILEINFO displays the owner and security information beneath the pathname for the file, because it occupies more space in the display than the corresponding Guardian security information

In the BRIEF display, the display heading line is identical to the one for Guardian files, because only one heading line appears in the display, although the display can include both types of files. The headings match the columns for Guardian files, not OSS files. Not all headings apply to OSS files.

The DETAIL display for OSS files also includes the creation date and the last open date. Other fields described for the DETAIL FILEINFO display for Guardian files do not apply to OSS files.

EXTENTS Display

This figure shows the EXTENTS display.

<i>file-name</i>			<i>date-and-time</i>
EXTENT	# OF PAGES	STARTING PAGE	[PART]
<i>extent-num</i>	<i>num-pages</i>	<i>start-page</i>	[name]
...			

The first line in the EXTENTS display is the same as the first line in the DETAIL display for tables, indexes, and Guardian files. The other fields are:

<i>extent-num</i>	The ordinal extent number of the entry. The first extent is 0. NONE indicates no extents are allocated.
<i>num-pages</i>	The number of disk pages (2048-byte units) in the extent.
<i>start-page</i>	The absolute page address of the first page of the extent.
<i>name</i>	For partitioned files, the partition name associated with the entry.

STATISTICS Display

This figure shows the STATISTICS display.

LEVEL	TOTAL BLOCKS	TOTAL RECS	AVG # RECS	AVG SLACK	AVG % SLACK	PART
1	1	1	1.0	4065	99	\$VOL1
DATA	1	1	1.0	4046	99	
FREE	0					
BITMAP	1					

The fields in the STATISTICS display are:

LEVEL	Indicates the tree level of the entry:
DATA	Data level (this is the only level shown for relative and entry-sequenced files)
<i>n</i>	Number of level with 1 as lowest level (for key-sequenced files)
	The remainder of the fields in the line apply to the indicated level.
TOTAL BLOCKS	The total number of blocks in use.
TOTAL RECS	The total number of records. At the DATA level, TOTAL RECS is the total number of data records in the file.
AVG # RECS	The average number of records per block.
AVG SLACK	The average number of unused bytes per block.
AVG % SLACK	The average percent of unused bytes for each block.
PART	For partitioned files, the volume name of the partition associated with the entry.
FREE TOTAL BLOCKS	For key-sequenced files, the total number of unused blocks from the beginning to the EOF.
FREE TOTAL RECS	For relative files, the total number of empty records from the beginning to the EOF.
BITMAP TOTAL BLOCKS	For relative and key-sequenced files only, the number of bitmap blocks.

Examples—FILEINFO

For these examples, assume that the subvolume \$VOL1.INVENT contains only catalog files and these files (see if this is ok), and that all SQL objects are defined in the catalog \$VOL1.INVENT:

ESTABLE	An unpartitioned, entry-sequenced table
PARTLOC	A partitioned, key-sequenced table
PARTLOC I	An index defined on PARTLOC

PVIEW A partitioned protection view that depends on PARTLOC
 PROG12 An SQL object program file
 SVIEW A shorthand view

- This command displays information for all the objects on \$VOL1.INVENT (including the catalog tables) in the default BRIEF format:

```
>> FILEINFO $VOL1.INVENT.*;

      CODE   EOF   LAST MODIF OWNER RWEP   TYPE  REC  BLOCK
$VOL1.INVENT
BASETABS  572A 12288 3Jan93 11:53 1,205 NCNC  K Ta   90  4096
COLUMNS 575A 32768 3Jan93 11:53 1,205 CCCC  K Ta  206  4096
COMMENTS 582A 12288 3Jan93 11:53 1,205 CCCC  K Ta  202  4096
CONSTRNT 580A    0 3Jan93 11:52 1,205 CCCC  K Ta 3068  4096
ESTABLE   10   8192 1Feb93 12:30 1,74  GUUU  E Ta   80  2048
...
PARTLOC  O   A 16384 1Feb93  9:45 1,74  GUUU PK Ta   12  4096
PARTLOC I O   A 16384 1Feb93  9:45 1,74  GUUU  K In   14  4096
PROG12   100P 19386 9Jan93 14:12 1,74  NUNU   Pg
PROGRAMS 581A 12288 3Jan93 11:53 1,205 CCCC  K Ta 3096  4096
PVIEW                    1,74  NNUU  P PVi
SVIEW                    1,74  GUUU   SVi
...
```

- This command displays DETAIL information for the file PARTLOC:

```
>> FILEINFO $VOL1.INVENT.PARTLOC, DETAIL;

$VOL1.INVENT.PARTLOC                               13 Jun 1995, 12:32
  SQL BASE TABLE
  CATALOG $VOL1.INVENT
  VERSION 1
  TYPE K
  FORMAT 1
  EXT ( 16 PAGES, 64 PAGES, MAXEXTENTS 160 )
  REC 10
  PACKED REC 9
  BLOCK 4096
  KEY ( COLUMN 0, OFFSET 0, LENGTH 3, ASC,
        COLUMN 1, OFFSET 4, LENGTH 2, ASC )
  PART ( 0, $VOL1, 16 PAGES, 64 PAGES, MAXEXTENTS 160, FORMAT 1, ([0],0) )
  PART ( 1, $WHS2, 16 PAGES, 64 PAGES, MAXEXTENTS 160, FORMAT 1, ("G00",0) )
  AUDIT
  BUFFERED
  AUDITCOMPRESS
  OWNER 1,74
  SECURITY (RWEP): GUUU
  MODIF: 12 Jun 1995, 20:12
  CREATION DATE: 10 Jun 1995, 20:01
  REDEFINITION DATE: 12 Jun 1995, 20:01
  LAST OPEN: 13 Jun 1995, 18:44
  EOF 12288 (0.1% USED)
  EXTENTS ALLOCATED: 1
  INDEX LEVELS: 1
  PARTITION ARRAY EXTENDED
```

If you request the STATISTICS display, the additional information appears below EXTENTS ALLOCATED instead of the INDEX LEVELS:

LEVEL	TOTAL BLOCKS	TOTAL RECS	AVG # RECS	AVG SLACK	AVG % SLACK	PART
1	1	1	1.0	4065	99	\$VOL1
DATA	1	15	15.0	3899	95	
FREE	0					
BITMAP	1					
1	1	1	1.0	4065	99	\$WHS2
DATA	1	31	31.0	3723	89	

```

FREE          0
BITMAP        1

```

- This command displays EXTENTS information about PARTLOC:

```

>> FILEINFO $VOL1.INVENT.PARTLOC, EXTENTS;
$VOL1.INVENT.PARTLOC          30 Oct 1994, 11:00
  EXTENT  # OF PAGES  STARTING PAGE  PART
      0         16         71199    $VOL1
      0         16        141363    $WHS2

```

- This command displays BRIEF information about collations by specifying the file attribute “COLLATION” in the WHERE clause of the qualified fileset list for FILEINFO:

```

>> FILEINFO \A.$A.A.* WHERE COLLATION;
      CODE   EOF   LAST MODIF   OWNER RWEP TYPE REC BLOCK
\A.$A.A
CASEINS   941  12288  16NOV93  11:53 175,213NONO K   3004  4096
ESPAÑOL   941  12288  17NOV93   9:10 175,213NONO K   3004  4096
FRANCAIS  941  12288  15NOV93  17:40 255,255NONO K   3004  4096
FRENCH    941  12288  16NOV93   8:20 175,213NONO K   3004  4096

```

- This FILEINFO command displays DETAIL information for a collation:

```

>> FILEINFO \A.$A.A.COL1, DETAIL;
\A.$A.A.COL1          1 Jun 1995,  9:51
  SQL CHARACTER PROCESSING RULES OBJECT
  CATALOG $A.A
  VERSION 300
  TYPE K
  FORMAT 1
  CODE 941
  EXT ( 16 PAGES, 64 PAGES, MAXEXTENTS 160 )
  REC 3004
  PACKED REC 3004
  BLOCK 4096
  KEY (COLUMN 0, OFFSET 0, LENGTH 2, ASC)
  AUDIT
  BUFFERED
  AUDITCOMPRESS
  OWNER 175,213
  SECURITY (RWEP): CCCC
  MODIF: 1 Jan 1992, 7:55
  CREATION DATE: 1 Jan 1992, 7:55
  REDEFINITION DATE: 1 Jan 1992, 7:55
  LAST OPEN: 1 Jan 1992, 7:55
  EOF 12288 (0.1% USED)
  EXTENTS ALLOCATED: 1
  INDEX LEVELS: 1

```

FILENAMES Command

FILENAMES is an SQLCI utility command that displays a set of file names that match a pattern specified with wild-card characters. You can restrict the list to objects described in specified catalogs.

```
FILENAMES [ qualified-fileset-list ] ;
```

qualified-fileset-list

specifies the files for which information is to be displayed. For more information, see [Qualified Fileset List](#) on page Q-1. If you omit *qualified-fileset-list*, FILENAMES displays the files on the current default subvolume.

If SMF is installed on your node, *qualified-fileset-list* can include files on `$.ZYS*` subvolumes. If you specify FROM CATALOG(S), all specified catalogs must be either logical or direct files.

FILENAMES displays file names on the OUT file, which is typically your terminal. For OSS files, FILENAMES displays ZYQ names.

Example—FILENAMES

This example lists all files in subvolumes that begin with the letter W, have file names that begin with the letters SQ, have the number 2 in the fifth character position, and are exactly five characters long:

```
>> FILENAMES W*.SQ??2;
```

To restrict the list to files created after December 31, 1994, and owned by user 12 of group 48, enter:

```
>> FILENAMES W*.SQ??2 WHERE CREATIONTIME AFTER DEC 31 1994
+> AND OWNER = 48,12;
```

FILES Command

FILES is an SQLCI utility command that displays the names of files that are on one or more subvolumes.

```
FILES [ subvol-spec ] ;
      [ ( subvol-spec [ , subvol-spec ] ... ) ]
```

subvol-spec

is the name of a Guardian subvolume or a name with wild-card characters that matches the names of several Guardian subvolumes.

If you do not specify any *subvol-spec*, FILES displays the files on the current default subvolume.

subvol-spec can include these wild-card characters in any part of the name except the node name:

- * matches zero or more characters
- ? matches any single character

FILES displays file names on the OUT file, which is typically your terminal. For OSS files, FILES displays ZYQ names.

Example—FILES

This example lists all files in volumes with names that end with the letter M and subvolumes on those volumes that have names that are six-characters long beginning with Z, with 00 in the fifth and sixth character positions:

```
>> FILES $*M.Z????00;
$SYSTEM.ZLOG00
      ZZEVO011  ZZEVO012  ZZEVO013  ZZEVO014  ZZEVCNF
>>
```


FILES Table

The FILES table is a catalog table that describes the attributes of files that contain tables and indexes. [Table F-2](#) describes the contents of the FILES table.

Table F-2. The FILES Table (page 1 of 2)

Column Name	Data Type	Description
1 FILENAME *	CHAR(34)	File name (same name as table, index, or partition in file)
2 FILETYPE	CHAR(1)	E if entry-sequenced R if relative K if key-sequenced
3 BLOCKSIZE	SMALLINT SIGNED	Block size in bytes: 512, 1024, 2058, or 4096
4 PRIMARYEXT	SMALLINT UNSIGNED	Size of primary extent in units of 2 KB. Superseded by PRIMARYEXT2
5 SECONDARYEXT	SMALLINT UNSIGNED	Size of secondary extent in units of 2 KB. Superseded by SECONDARYEXT2
6 MAXEXTS	SMALLINT SIGNED	Maximum number of extents in file (both primary and secondary)
7 LOCKLENGTH	SMALLINT SIGNED	Locklength file attribute; 0 if same as primary key
8 PARTITIONED	CHAR(1)	Y if partitioned N if not
9 AUDIT	CHAR(1)	Y if audited by TMF N if not
10 DCOMPRESS	CHAR(1)	Y if data keys compressed N if not
11 ICOMPRESS	CHAR(1)	Y if index keys compressed N if not
12 CLEARONPURGE	CHAR(1)	Y if all data in file is physically erased from disk when file purged N if not
13 SERIALWRITES	CHAR(1)	Y if serial mirror writes N if parallel
14 VERIFIEDWRITES	CHAR(1)	Y if disk read follows each disk write N if not
15 BUFFERED	CHAR(1)	Y if writes to disk are buffered N if not
16 NOPURGEUNTIL	LARGEINT SIGNED	Julian timestamp of earliest time file can be purged; 0 if file can be purged any time

Table F-2. The FILES Table (page 2 of 2)

Column Name	Data Type	Description
17 EOF	INTEGER SIGNED	Relative byte address of first unused byte of last block in partition or file; updated by UPDATE STATISTICS. Superseded by EOF2.
18 NONEMPTYBLOCKCOUNT	LARGEINT SIGNED	Number of blocks that contain at least one row of data; updated by UPDATE STATISTICS
19 RECORDSIZE	SMALLINT SIGNED	Maximum length of packed record
20 AUDITCOMPRESS	CHAR(1)	Y if AUDITCOMPRESS N if not
21 PARTITIONARRAY	CHAR (30)	STANDARD if the type of partition array is that used by default EXTENDED if the type of partition array is extended and supports additional partitions FORMAT2ENABLED if the type of partition array supports both FORMAT 1 and FORMAT 2 partitions
22 DCOMPRESSTYPE	CHAR(1)	1 if DCOMPRESS 2 if DCOMPRESS2
23 PRIMARYEXT2	INTEGER UNSIGNED	Size of primary extent in units of 2 KB
24 SECONDARYEXT2	INTEGER UNSIGNED	Size of secondary extent in units of 2 KB
25 EOF2	LARGEINT SIGNED	Relative byte address of first unused byte of last block in partition or file; updated by UPDATE STATISTICS
26 FILEFORMAT	CHAR(1)	Format of partition: 1 if FORMAT 1 2 if FORMAT 2

* Indicates primary key

The columns FILENAME through RECORDSIZE (1 through 19) were created in version 1. The column AUDITCOMPRESS (20) was added in version 2. The column PARTITIONARRAY (21) was added in version 320. Column DCOMPRESSTYPE was added in version 345. Columns PRIMARYEXT2, SECONDARYEXT2, and EOF2 were added in version 350 and supersede PRIMARYEXT, SECONDARYEXTENT, and EOF.

Guardian names in the FILES table are fully qualified and use uppercase characters.

Filesets

A fileset is a set of objects and files specified as a Guardian name that optionally includes these wild-card characters in the *volume*, *subvolume*, or *file-id* portions of the name:

- ? matches any single character.
For example, TBL? matches TBL1 or TBLX but not TBL48.
- * matches any 0 to 8 characters.
For example, * matches any 0 to 8-character name.

VOL matches NEWVOL, OLDDVOL1, VOL45, and so forth.

Notice that a single Guardian name that includes wild-card characters can represent a fileset that includes many files. You cannot use a wild-card in the node portion of a Guardian name that specifies a fileset.

You can use a DEFINE to specify a fileset, but you cannot use wild-card characters in the DEFINE name or in the file name you specify on the DEFINE. As a result, a fileset you specify with a DEFINE always consists of a single object or file.

Many SQLCI commands allow you to use filesets or fileset lists to specify a set of tables and files for the command. A fileset list can be a simple fileset list or a qualified fileset list.

A simple fileset list is a list of one or more filesets in this form:

```
{ fileset
{ ( fileset [ , fileset ] ... ) }
```

A qualified fileset list can include clauses that restrict the files in the set in various ways. For information, see [Qualified Filesset List](#) on page Q-1.

Examples—Filesets

- This example on FILEINFO requests information about the fileset that includes all files on subvolume \NY.\$HDQTR.HR:

```
FILEINFO \NY.$HDQTR.HR.*;
```

- This PURGE command purges the fileset that includes all files on subvolume SV3 with names that end in X:

```
PURGE SV3.*X;
```

FORMAT File Attribute

FORMAT is a file attribute that specifies the format of a partition. A Format 1 partition is limited to one gigabyte. A Format 2 partition might grow to one terabyte, or the size of a single disk volume.

```
{ FORMAT 1 | FORMAT 2 }
```

If the table's PARTITION ARRAY VALUE has been set to STANDARD or EXTENDED, the default format is FORMAT 1. If the table's PARTITION ARRAY VALUE has been set to FORMAT2ENABLED, the default format is FORMAT 2.

Considerations—FORMAT

- FORMAT specifies the format for the base partition of the table or index. To specify the format for secondary partitions, you must specify FORMAT in the PARTITION clauses defining each partition.
- If you specify FORMAT 2 and do not use the PARTITION ARRAY clause, the PARTITION ARRAY is set to FORMAT2ENABLED.
- Only key-sequenced tables can be Format 2 enabled.
- To create Format 2 partitions, a file's PARTITION ARRAY VALUE must be FORMAT2ENABLED.

FREE RESOURCES Statement

FREE RESOURCES is a DCL statement that releases locks, closes cursors, and flushes insert/update buffers for audited and, optionally, nonaudited objects.

The exact effect of FREE RESOURCES depends on how the locks were acquired, whether the affected objects are audited or nonaudited, and whether a transaction is in progress, as explained later in this entry.

You can use FREE RESOURCES instead of explicit UNLOCK TABLE or CLOSE CURSOR statements. COMMIT WORK and ROLLBACK WORK automatically perform all the operations of FREE RESOURCES.

```
FREE RESOURCES [ AUDITONLY ] | [ CLOSE TABLES ]
```

AUDITONLY

directs SQL to retain existing locks on nonaudited objects.

CLOSE TABLES

directs SQL to close tables. Otherwise, users or programs without access privileges can open them.

Considerations—FREE RESOURCES

- If CONTROL TABLE SEQUENTIAL INSERT/UPDATE is set to ENABLE or ON, FREE RESOURCES also flushes the insert/update buffer for audited and nonaudited tables.
- On audited tables, FREE RESOURCES releases locks acquired only within the current TMF transaction using STABLE access. FREE RESOURCES releases locks that apply only to data being read. Locks on data that has been inserted, updated, or deleted by the current transaction are released automatically when the transaction completes.

FREE RESOURCES does not affect a table lock on an audited table. The lock is released when the transaction ends unless SQL acquired the lock under STABLE access and no updates were performed.

- On nonaudited tables, unless you specify AUDITONLY, FREE RESOURCES releases all locks acquired by DML statements with REPEATABLE access, whether or not a TMF transaction is in progress.

You must use either FREE RESOURCES or UNLOCK TABLE to release locks on nonaudited tables under these conditions:

- A DML statement with REPEATABLE access operates on nonaudited tables or on views with underlying nonaudited tables.
- LOCK TABLE statements are in effect.
- A host program is returning control to a requester but holds locks on nonaudited files because it used the AUDITONLY option on COMMIT WORK, ROLLBACK WORK, or another FREE RESOURCES statement.

You must use FREE RESOURCES or COMMIT WORK to ensure the insert/update buffer is flushed for nonaudited tables before you exit SQLCI or a host program. (In programs, you must then check SQLCA for flush errors.)

- FREE RESOURCES closes all cursors on nonaudited tables or views unless you specify the AUDITONLY option.

FREE RESOURCES closes all cursors with BROWSE access to audited tables or views. FREE RESOURCES closes a cursor with STABLE or REPEATABLE access to an audited table only if the FREE RESOURCES statement executes in the same transaction as the most recent OPEN of the cursor.

Examples—FREE RESOURCES

- This example shows FREE RESOURCES used to release locks acquired by SQLCI on a nonaudited file.

Suppose that JOB is a nonaudited table and AUTOWORK AUDITONLY is ON. Because the INSERT uses REPEATABLE access, locks are not released when the

SQLCI-defined transaction ends. The FREE RESOURCES statement (without AUDITONLY) releases the locks.

```
>> VOLUME $VOL1.PERSNL;
>> INSERT INTO JOB VALUES (650, "ADMIN ASSISTANT")
+> REPEATABLE ACCESS;
-- 1 row(s) inserted.
...
>> FREE RESOURCES;
-- SQL operation complete.
```

- This example shows FREE RESOURCES in a host program where it also deallocates buffer space used for cursors:

```
EXEC SQL OPEN CURSOR1;
...
EXEC SQL FETCH CURSOR1;
...
EXEC SQL DELETE FROM...
      WHERE CURRENT OF CURSOR1;
...
EXEC SQL FREE RESOURCES;
```

Functions

This table summarizes the functions in NonStop SQL/MP:

<u>AVG Function</u>	Returns the average of a set of numbers
<u>CAST Function</u>	Associates a data type with a parameter
<u>COMPUTE_TIMESTAMP Function</u>	Returns a Julian timestamp for a specified date and time
<u>CONVERTTIMESTAMP Function</u>	Converts a Julian timestamp to a DATETIME value
<u>COUNT Function</u>	Counts the rows returned from a query or the distinct values in a column
<u>CURRENT Function</u>	Returns the current date and time
<u>CURRENT_TIMESTAMP Function</u>	Returns a Julian timestamp for the current date and time
<u>DATEFORMAT Function</u>	Formats a date-time value
<u>DAYOFWEEK Function</u>	Returns an integer that represents a day of the week
<u>EXTEND Function</u>	Adjusts the range of fields for a data-time value
<u>JULIANTIMESTAMP Function</u>	Converts a date-time value to a Julian timestamp
<u>LINE_NUMBER Function</u>	Returns the line number of the current detail line in a report
<u>MAX Function</u>	Returns a maximum value for a column or set of values

MIN Function	Returns a minimum value for a column or set of values
PAGE_NUMBER Function	Returns the page number of the current page in a report
SETSCALE Function	Specifies the scale of a host variable
SUM Function	Computes the sum of a set of numbers
UPSHIFT Function	Upshifts single-byte characters

For more information, see the entry for a specific function.

FUP Command

FUP is an SQLCI command that executes File Utility Program (FUP) commands. FUP commands perform operations such as creating, purging, and displaying files. (For a complete description of FUP commands, see the *File Utility Program (FUP) Reference Manual*.)

```
FUP [ /run-option-list/ ] [fup-command-line] ;
```

run-option-list

is one or more run options of the TACL RUN command, described in the *TACL Reference Manual*. If you do not specify IN *in-file* or OUT *out-file*, the SQLCI IN file and OUT file are used. If the current SQLCI OUT file is a disk file, you must close the file by entering OUT before you use the FUP command. SQLCI cannot redirect FUP output to a disk file.

fup-command-line

is a FUP command up to 132 characters long. To continue a command on the next line, press Return. The SQLCI command continuation prompt (+>) appears.

If you specify *fup-command-line*, FUP returns you to SQLCI after the command finishes. If you omit *fup-command-line*, FUP prompts you for each command and returns you to SQLCI when you enter the EXIT command.

FUP Commands and SQL Objects

FUP commands work on SQL-compiled object program files but generally do not work on other SQL objects. Some SQLCI commands perform the same or similar operations on SQL objects. For example, you can use FILEINFO or FUP INFO with either SQL objects or Enscribe files, but you must use SQLCI's COPY, DUP, LOAD, PURGE, or SECURE to operate on SQL objects.

[Table F-3](#) on page 34 lists the FUP commands that perform operations on files and indicates whether the commands work with objects other than SQL object programs. For FUP commands that do not work with SQL objects, [Table F-3](#) on page F-34 lists the SQL command or SQL utility that performs the equivalent function if one exists.

Table F-3. FUP Commands and SQL Objects

FUP Command	Works on SQL Objects	Equivalent SQL Command or Utility
FUP ALLOCATE	No	ALTER CATALOG, ALTER INDEX, ALTER PROGRAM, ALTER TABLE, ALTER VIEW
FUP ALTER	No	ALTER CATALOG, ALTER INDEX, ALTER PROGRAM, ALTER TABLE, ALTER VIEW
FUP BUILDKEYRECORDS	No	
FUP CHECKSUM	Yes (only nonaudited)	
FUP COPY	No	COPY utility
FUP CREATE	No	CREATE TABLE, CREATE INDEX
FUP DEALLOCATE	No	ALTER CATALOG, ALTER INDEX, ALTER PROGRAM, ALTER TABLE, ALTER VIEW
FUP DUP	No	DUP utility
FUP FILES	Yes	
FUP GIVE	No	ALTER CATALOG, ALTER INDEX, ALTER PROGRAM, ALTER TABLE, ALTER VIEW, SECURE
FUP INFO	Yes	
FUP LICENSE	Yes	
FUP LISTLOCKS	Yes	
FUP LISTOPENS	Yes (programs only)	
FUP LOAD	No	LOAD
FUP LOADALTFILE	No	LOAD
FUP PURGE	Yes (programs only)	DROP, PURGE
FUP PURGEDATA	No	PURGEDATA
FUP RELOAD	Yes	
FUP RENAME	No	
FUP REVOKE	No	
FUP SECURE	No	ALTER CATALOG, ALTER INDEX, ALTER PROGRAM, ALTER TABLE, ALTER VIEW, SECURE

You can also use ALLOW, CTRL-Y, EXIT, FC, HELP, RESET, SET, SHOW, SYSTEM, and VOLUME to perform operations on files. When you return to SQLCI, the default system and volume are those that were in effect when you invoked FUP.

If a table or view is encountered during the processing of a FUP command that does not support SQL objects, the table or view is skipped and a warning message is issued.

Considerations—FUP

- If you use FUP DUP to duplicate an object program, the SQL SENSITIVE and SQL VALID flags in the file label of the new file are turned off, and you receive a warning message that the file must be compiled by SQL.
- If you apply the FUP LICENSE command to an SQL object program file, the file is licensed. This method is the only way to license an SQL object program file.
- You can display information about Enscribe files and SQL objects by using either the FUP INFO command or the SQL FILEINFO utility. The only significant difference between the two utilities is that the *fileset-list* parameter of FUP INFO does not support the FROM CATALOG option and does not handle wild-card characters in the same way as the *qualified-fileset-list* parameter of the SQL FILEINFO utility.

Examples—FUP

- This example displays the files on the current subvolume:

```
>> FUP FILES;
$VOL1.DFLT
A1          A2          A3          A4
```

- This example lists processes that currently have files and objects open on subvolume ZYQ00001:

```
>> FUP LISTOPENS ZYQ00001.*;

$OSS000.ZYQ00001.Z00000T6
  PID      MODE  USERID  SD  MYTERM      PROGRAM FILE NAME
000,00,0000  W   -S 104,2   00  $ZTN.#PTY03  $VOL1.S.PROG

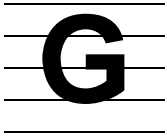
$OSS000.ZYQ00001.Z00005NM
  PID      MODE  USERID  SD  MYTERM      PROGRAM FILE NAME
175,02,0111  R   -S 104,11  01  $ST.#PTY04   $COBOL.COBOL.COBOL
>>
```

- This example lists detailed information about an OSS file:

```
>> FUP INFO $OSS001.ZYQ00002.Z00000A1,DETAIL

$OSS001.ZYQ00002.Z00000A1
OSS
```

```
PATH: /lt2/br/src/csrc/lex.o
OWNER 104,2
SECURITY: -rw-rw-rw-
CREATION DATE: 7 Feb 1995, 13:55
ACCESS TIME: 7 Feb 1995, 13:55
EOF 286084
```



Generalized Owner

A generalized owner of an SQL object or Guardian file is any user ID that has ownership privileges for the file.

On the node where the file is located, the generalized owner includes the user ID that owns the file, the group manager of the group that includes that user ID, and the super ID. If the owner can purge the file from another node in the network (as specified with the fourth character of the security string), the generalized owner also includes the same user ID on other nodes, the group manager on other nodes, and the super ID on other nodes.

For more information, see [Security](#) on page S-11.

GET CATALOG OF SYSTEM Statement

GET CATALOG OF SYSTEM is a DSL statement that returns the name of a local or remote system catalog.

```
GET CATALOG OF SYSTEM [ \node ] [ INTO :var ]
```

node

is the name of a node. The default is the local node.

INTO :var

(static SQL programs only) specifies a host variable in which to return the system catalog name.

Considerations—GET CATALOG OF SYSTEM

In host language programs, GET CATALOG OF SYSTEM returns the fully qualified system catalog name as 25 characters, left-justified, and padded with blanks. The variable to receive the name must be compatible with the SQL data type CHAR(25).

For static SQL, specify the variable in the INTO clause of GET CATALOG. For dynamic SQL, specify the variable in the RETURNING clause of the EXECUTE or specify an output SQLDA in the RETURNING USING DESCRIPTOR clause of the EXECUTE.

GET CATALOG OF SYSTEM also sets SQLCODE to indicate status and fills in the SQLCA. GET CATALOG OF SYSTEM has no EXPLAIN output.

Examples—GET CATALOG OF SYSTEM

- This SQLCI example shows how GET CATALOG OF SYSTEM returns a catalog name in an SQLCI session:

```
>>GET CATALOG OF SYSTEM \SYSA;  
CATALOG: \SYSA.$SYSTEM.SQL  
--- SQL operation complete.
```

- This static SQL example from a C, Pascal, or TAL program retrieves the system catalog name for \SYSA and stores it in hostvar1:

```
EXEC SQL GET CATALOG OF SYSTEM \SYSA INTO :hostvar1;
```

- This static SQL example from a COBOL program retrieves the system catalog name for the local system and stores it in HOSTVAR2:

EXEC SQL GET CATALOG OF SYSTEM INTO :HOSTVAR2 END-EXEC.

- This example from a COBOL program use dynamic SQL to retrieve the system catalog name for \SYSA and store it in a host variable, HOSTVAR3:

```
MOVE "GET CATALOG OF SYSTEM \SYSA " TO STRING1.
EXEC SQL PREPARE S1 FROM :STRING1 END-EXEC.
EXEC SQL EXECUTE S1 RETURNING :HOSTVAR3 END-EXEC.
```

- This example from a COBOL program use dynamic SQL to retrieve the system catalog name for \SYSA and store it in MYSQLDA, an output SQLDA:

```

MOVE "GET CATALOG OF SYSTEM \SYSA " TO STRING2.
EXEC SQL PREPARE S2 FROM :STRING2 END-EXEC.
EXEC SQL EXECUTE S2 RETURNING USING DESCRIPTOR :MYSQLDA
      END-EXEC.

```

GET VERSION Statement

GET VERSION is a DSL statement that returns the version of an SQL catalog or object or the version of the SQL/MP software on a node.

```

GET VERSION OF { { CATALOG      } s }
                { { COLLATION    } }
                { { INDEX        } name }
                { { TABLE       } } [ INTO :var ]
                { { VIEW         } }
                { { SYSTEM [ name ] } }

```

CATALOG, COLLATION, INDEX, TABLE, VIEW, or SYSTEM

specifies the type of item for which to return the version.

name

is the name of the item for which to return the version.

For type CATALOG, *name* must be a catalog name or a DEFINE of class CATALOG. For type SYSTEM, *name* is optional (the default is the local node) but must be a node name with a leading “\” if specified. For all other types *name* must be a Guardian name (or an equivalent DEFINE).

INTO :*var*

(static SQL programs only) specifies a host variable in which to return the version.

Consideration—GET VERSION

In host language programs, GET VERSION returns an integer value. The variable to receive the value must be compatible with the SQL data type UNSIGNED SMALLINT.

For static SQL, specify the variable in the INTO clause of GET VERSION. For dynamic SQL, specify the variable in the RETURNING clause of the EXECUTE or specify an output SQLDA in the RETURNING USING DESCRIPTOR clause of the EXECUTE. (You cannot use GET VERSION with EXECUTE IMMEDIATE.)

GET VERSION also sets SQLCODE to report status and fills in the SQLCA. GET VERSION has no EXPLAIN output.

Examples—GET VERSION

- This SQLCI example retrieves the version of table mytable:

```
GET VERSION OF TABLE mytable;
VERSION: 1
--- SQL operation complete.
```

- This SQLCI example retrieves the version of catalog mycat:

```
GET VERSION OF CATALOG mycat;
VERSION: 300
--- SQL operation complete.
```

- This SQLCI statement retrieves the version of index myindx:

```
GET VERSION OF INDEX myindx;
VERSION: 315
---SQL operation complete.
```

- This static SQL statement from a C, Pascal, or TAL program retrieves the version of table mytable and stores it in hostvar1:

```
EXEC SQL GET VERSION OF TABLE mytable INTO :hostvar1;
```

- This static SQL statement from a COBOL program retrieves the version of index IDX2 and stores it in hostvar2:

```
EXEC SQL GET VERSION OF INDEX idx2 INTO :hostvar2;
```

- This COBOL example uses dynamic SQL to retrieve the version of catalog \sysa.\$vol1.mycat and store it in an SQLDA:

```
MOVE "GET VERSION OF CATALOG \sysa.$vol1.mycat" TO string1.
EXEC SQL PREPARE s1 FROM :string1 END-EXEC.
EXEC SQL EXECUTE s1
      RETURNING USING DESCRIPTOR :mysqlda END-EXEC.
```

GET VERSION OF PROGRAM Statement

GET VERSION OF PROGRAM is a DSL statement that returns the program catalog version (PCV), program format version (PFV), or host object SQL version (HOSV) of an SQL program that is registered in an SQL catalog.

```
GET { CATALOG | FORMAT | HOST OBJECT }
    VERSION OF PROGRAM program [ INTO :var ]
```

```
{ CATALOG | FORMAT | HOST OBJECT }
```

specifies the type of program version to return: PCV (CATALOG), PFV (FORMAT), or HOSV (HOST OBJECT).

program

is the Guardian name of the program for which to return a version. (If the program is an SQL program in an OSS file, *program* must be the Guardian-format ZYQ name of the program file, not the pathname.)

For a CATALOG or FORMAT version, *program* must be an existing program that has been SQL-compiled. For a HOST OBJECT version, *program* must be an existing program that has been host language-compiled (and might have been SQL-compiled).

```
INTO :var
```

(static SQL programs only) specifies a host variable in which to return the version number.

Consideration—GET VERSION OF PROGRAM

In host language programs, GET VERSION OF PROGRAM returns an integer value. The variable to receive the value must be compatible with the SQL data type UNSIGNED SMALLINT.

For static SQL, specify the variable in the INTO clause of GET VERSION OF PROGRAM. For dynamic SQL, specify the variable in the RETURNING clause of the EXECUTE or specify an output SQLDA in the RETURNING USING DESCRIPTOR clause of the EXECUTE. (You cannot use GET VERSION OF PROGRAM with EXECUTE IMMEDIATE.)

GET VERSION OF PROGRAM also sets SQLCODE to report status and fills in the SQLCA. GET VERSION OF PROGRAM has no EXPLAIN output.

Examples—GET VERSION OF PROGRAM

- This SQLCI example retrieves the PCV, PFV, and HOSV of program MYPROG:

```
>>GET CATALOG VERSION OF PROGRAM myprog;
VERSION: 315
--- SQL operation complete.
>>GET FORMAT VERSION OF PROGRAM myprog;
VERSION: 315
--- SQL operation complete.
>>GET HOST OBJECT VERSION OF PROGRAM myprog;
VERSION: 315
--- SQL operation complete.
>>
```

- This static SQL statement from a C, Pascal, or TAL program retrieves the PCV of program MYPROG and stores it in hostvar1:

```
EXEC SQL GET CATALOG VERSION OF PROGRAM myprog
      INTO :hostvar1;
```

- This static SQL statement from a COBOL program retrieves the HOSV of the program MYPROG and stores it in HOSTVAR2:

```
EXEC SQL GET HOST OBJECT VERSION OF PROGRAM MYPROG
      INTO :HOSTVAR2 END-EXEC.
```

- This COBOL example uses dynamic SQL to retrieve the PCV of the program MYPROG and store it in HOSTVAR3:

```
MOVE "GET CATALOG VERSION OF PROGRAM MYPROG " TO STRING1.
EXEC SQL PREPARE s1 FROM :STRING1 END-EXEC.
EXEC SQL EXECUTE s1 RETURNING :HOSTVAR3 END-EXEC.
```

- This C example uses dynamic SQL to retrieve the HOSV of the program MYPROG and store it in an output SQLDA:

```
string2 = "GET HOST OBJECT VERSION OF PROGRAM MYPROG ";
EXEC SQL PREPARE s2 FROM :string2;
EXEC SQL EXECUTE s2 RETURNING USING DESCRIPTOR :mysqlda;
```

GOAWAY Command

GOAWAY is a TACL utility program that allows a user with super ID authority to delete Guardian SQL files or shadow labels that cannot be removed with other commands or utilities.

You execute GOAWAY from TACL with this command:

```
GOAWAY [ /IN cmdfile/ ] [ filename[:S] ] ;
```

/IN cmdfile/

specifies an EDIT file that lists SQL objects, programs, or shadow labels to delete. List one item per line, using syntax described for the *filename* [:S] option.

filename[:S]

is the name (or equivalent DEFINE) of an SQL object or program to delete, or the name of a shadow label to delete.

The optional :S suffix specifies a shadow label.

If SMF is installed on your node, *filename* [:S] must not specify an object, program, or shadow label on a \$*.ZYS*. subvolume.

Considerations—GOAWAY

- The GOAWAY utility deletes files or file labels but does not delete their corresponding catalog entries.
- If you enter GOAWAY without options, GOAWAY prints instructions and prompts for *filename* [:S] entries.
- GOAWAY must be licensed and can be used only by the super ID.
- Do not use the GOAWAY utility as a substitute for DROP, PURGE, or CLEANUP operations. Misuse of the GOAWAY utility can corrupt files.

Examples—GOAWAY

- This example deletes the table mytable:
42> GOAWAY mytable
- This example deletes the shadow label indexb:
43> GOAWAY indexb:S

Group Manager

A group manager is a Guardian user ID that has user number 255. By convention, a group manager also has the user name MANAGER, but this is not required.

32,255	Group manager user ID number
DP.MANAGER	Typical group manager user ID name

A group manager can act as the owner of any object or file on the local node owned by a member of the Guardian security group to which the group manager belongs (group number 32 or group name DP in the examples just listed). A group manager can also act as the owner of an object or file owned by a member of the group on a remote node, provided the file is secured so that the owner has purge authority on remote nodes. The group manager is said to be a *generalized owner* of such objects and files.

For more information, see [Security](#) on page S-11.

Guardian Names

A Guardian name is a form of name used for disk files and other entities on the Guardian operating system.

NonStop SQL/MP uses Guardian names as names for SQL tables, views, indexes, partitions, collations, and programs. NonStop SQL/MP uses a portion of a Guardian name (the subvolume name) as an SQL catalog name.

[[\ <i>node</i> .] [\$ <i>volume</i> .] <i>subvol</i> .] <i>file-id</i>

\node

is the name of a node on a system. The name must be preceded by a backslash and consist of a letter followed by 1 to 6 letters or digits.

\$volume

is the name of a disk volume. The name must be preceded by a dollar sign and consist of a letter followed by 1 to 7 letters or digits.

subvol

is a subvolume name that consists of a letter followed by 1 to 7 letters or digits. A subvolume name, optionally preceded by a node and volume name but without this *file-id*, can be an SQL catalog name.

file-id

is the name of a Guardian disk file or the name of a SQL/MP table, view, index, partition, collation, or program. The name consists of a letter followed by 1 to 7

letters or digits. This portion of the name is sometimes called the “simple file name.”

Considerations—Guardian Names

If you do not fully qualify a Guardian name, SQL uses the current default node, volume, and subvolume names to expand the name as needed at name-resolution time. You can change the current defaults in effect for a program by changing the defaults in the process that executes the program or by setting the `=_DEFAULTS DEFINE` in the program. You can change the current defaults in SQLCI with the `VOLUME` or `CATALOG` commands or by setting the `=_DEFAULTS DEFINE`.

The time at which name resolution occurs depends on the statement in which a name is used and whether a `CONTROL QUERY BIND NAMES AT EXECUTION` directive was in effect at the time the statement was compiled or prepared (compiled with the `PREPARE` statement). For more information, see [Name Resolution](#) on page N-2 or [CONTROL QUERY Directive](#) on page C-74.

Example—Guardian Names

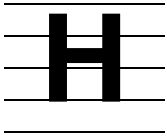
These are all Guardian names:

`ORDERS`

`SALES.ORDERS`

`$VOL1.SALES.ORDERS`

`\SYS1.$VOL1.SALES.ORDER`



HEADING Clause

HEADING is a clause in the ALTER TABLE, ALTER VIEW, CREATE TABLE, and CREATE VIEW statements that specifies a default heading for a column.

HEADING <i>string</i> NO HEADING

HEADING *string*

specifies a default heading for a column, expressed as a string of single-byte or multi-byte characters enclosed in single or double quotation marks. *string* can be 0 to 132 bytes long.

string cannot include the character string specifier normally allowed on a string literal.

To indicate line breaks in a heading, use the new-line character. The default new-line character is a slash (/). You can specify up to 50 lines in a single heading.

NO HEADING

specifies that no default heading should be printed for the column.

Consideration—HEADING

If you do not specify the HEADING clause, SQL uses the column name when printing or displaying the column.

Example—HEADING

This ALTER TABLE statement specifies *Customer* as the default heading for column CUSTOMER_NAME in table CUST. The new heading replaces any existing heading for the column.

```
ALTER TABLE CUST COLUMN CUSTOMER_NAME HEADING "Customer"
```

HEADINGS Option

HEADINGS is an option of the report writer SET STYLE command. HEADINGS activates or suppresses the printing of headings in the current report and in subsequent reports until you reset the HEADINGS option or end the SQLCI session.

Setting HEADINGS OFF is the same as specifying NOHEAD for every print item in your report.

For information about the way report writer determines headings, see the Considerations subsection in [DETAIL Command](#) on page D-47.

```
HEADINGS { ON }
          { OFF }
```

The default is ON.

Example—HEADINGS

To omit headings from reports, enter:

```
>> SET STYLE HEADINGS OFF;
```

HELP Command

HELP is an SQLCI command that displays information about SQL statements, SQLCI commands, and other SQL-related topics.

```
HELP { ALL }
     { help-topic [ , SYNTAX ] } ;
     {             [ , DETAIL ] }
     {             [ , EXAMPLE ] }
```

help-topic

is the topic you want information about.

SYNTAX, DETAIL, or EXAMPLE

specifies the type of information you want:

- | | |
|---------|--|
| SYNTAX | displays the syntax of a statement, command, language element, compiler directive, or a summary of a topic. |
| DETAIL | displays all available information about the topic (syntax or summary, examples, and any detailed text that is available). |
| EXAMPLE | displays examples of using the statement, command, language element, or compiler directive. (Available for all statements and commands, but not for all topics.) |

The default is SYNTAX.

Considerations—HELP

- HELP topics correspond to the main words of each major entry in the *SQL/MP Reference Manual*. For example, HELP is available for the topic CREATE TABLE (not CREATE or CREATE TABLE statement).

In some cases, HELP contains additional “pointer entries” that direct you to available topics. For example, if you enter HELP CREATE, SQLCI responds by listing available topics that begin with the word “create”:

```
Enter:  HELP create catalog
        create constraint
        create collation
        create index
        create system catalog
        create table
        create view
```

- If the entire help text for a topic fits on the screen, SQLCI displays the text and returns you to the standard SQLCI prompt. If the help text is too long for one screen, SQLCI displays the first part of the text followed by a continuation prompt. SQLCI displays information at the bottom of each screen of text that tells you what to press to display more detail or to return to the standard prompt.

Examples—HELP

- This command displays the available HELP topics:

```
HELP ALL;
```

- This commands display information about the CREATE TABLE statement. The information displayed by the third command includes all the information displayed by the first two commands:

```
HELP CREATE TABLE;
HELP CREATE TABLE, EXAMPLE;
HELP CREATE TABLE, DETAIL;
```

HELP TEXT Statement

HELP TEXT is a DDL statement that specifies help text for a column of a table or view.

```
HELP TEXT FOR COLUMN column ON { name }
    IS text-line [, text-line ] ...
```

column

is the name of a column with which to associate help text.

name

is the name of a table or view that includes the column.

text-line

is a line of help text in the form of a string of single-byte or multibyte characters enclosed in single or double quotation marks. The string can be 0 to 132 bytes long, but SQL issues a warning if it contains more than 77 bytes.

The set of strings you specify replaces any existing help text for the column. To delete help text, specify a null string ("").

Consideration—HELP TEXT

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 execute a HELP TEXT statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For information, see [DDL \(Data Definition Language\) Statements](#) on page D-20.

Example—HELP TEXT

This example adds help text for the EMPNAME column of the EMP table:

```
HELP TEXT FOR COLUMN EMPNAME ON EMP IS
    "NAME OF EMPLOYEE",
    "THE FORMAT IS LAST-NAME, FIRST-NAME, MI";
```

HISTORY Command

HISTORY is an SQLCI command that displays the commands or statements most recently entered during the SQLCI session. HISTORY identifies each command by a number that you can use with the FC command to reexecute or edit the command. (HISTORY is similar to the TACL command HISTORY.)

```
HISTORY [ number ] ;
```

The default is 10.

number

is the number of commands to display.

The history buffer contains at the most 25 commands. You can use the FC command to edit and reexecute a command in the history buffer, or use the exclamation point command (!) to reexecute a command without modifying it.

Example—HISTORY

This command displays the last five commands or statements entered during the SQLCI session:

```
>> HISTORY 5 ;
4> SHOW PREPARED * ;
5> VOLUME PERSNL ;
6> ENV ;
7> LOG ;
8> HISTORY 5 ;
```

Host Identifiers

Host identifiers are names used in host language programs to identify data items, structures, functions, or labels declared in the programs.

In an SQL statement or directive, a host identifier is always preceded by a colon (:), but other rules for host identifiers depend on the programming language. For information about the rules for host identifiers in a specific language, see the SQL/MP programming manual for the host language you use.

Host Programs

A host program or host language program is a program that contains both host-language statements and embedded SQL statements.

You can write SQL/MP host programs in C, COBOL, Pascal, or TAL. For more information, see one of these manuals:

SQL/MP Programming Manual for C

SQL/MP Programming Manual for COBOL

SQL Programming Manual for Pascal

SQL Programming Manual for TAL

C programs can run in the Guardian or OSS environments; other programs run only in the Guardian environment.

Host Variables

Host variables are data items declared in a host program and used in both host language statements and SQL statements. They provide for communication between SQL and the host language.

A host variable can be any valid host -language variable that has a corresponding SQL data type. You can include host variables in many SQL statements and in SQL expressions. The syntax for a host variable that appears in an SQL statement follows:

```
:host-identifier[ [ INDICATOR ] : indicator-host-identifier ]

[ TYPE AS { DATETIME [ start-dt TO ] end-dt } ]
[ { DATE | TIME | TIMESTAMP } ]
[ { INTERVAL start-dt } ]
[ { [ ( start-field-precision ) ] } ]
[ { [ TO end-dt ] } ]
```

host-identifier

is the name of the host variable as declared in the host program; *host-identifier* must conform to the naming rules of the host language.

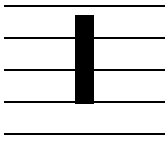
[INDICATOR] : *indicator-host-identifier*

specifies an indicator variable for handling null values returned to the host variable or inserting null values into the database through the host variable. For more information, see [Indicator Variables and Indicator Parameters](#) on page I-11.

TYPE AS

indicates that values in the host variable have a date-time or INTERVAL data type. (SQL interprets such values as character values unless you specify TYPE AS.)

For more information about declaring and using host variables, see the SQL/MP programming manual for your host language.



ICOMPRESS File Attribute

ICOMPRESS is a file attribute that controls key compression in index blocks. ICOMPRESS applies only to key-sequenced tables and indexes.

```
{ ICOMPRESS | NO ICOMPRESS }
```

The table default is NO ICOMPRESS.

The index default is the table value at index creation.

Considerations—ICOMPRESS

- Occasionally, the use of ICOMPRESS can reduce the number of index levels. Reducing the number of index levels improves performance.

You can check the number of index levels by looking at the INDEXLEVELS column of the INDEXES catalog table. (ALTER INDEX does not update the INDEXLEVELS column but UPDATE STATISTICS does.) If ICOMPRESS does not reduce index levels, it lowers performance but saves disk space.

- The disk process does a sequential scan of all indexes in the block, beginning at the start of the block. This function is similar to DCOMPRESS; savings result when the index contains many like values.
- Relative and entry-sequenced tables always have the NO ICOMPRESS attribute, but index block compression has no meaning for them.

IF/THEN/ELSE Clause

IF/THEN/ELSE is an SQLCI report writer clause that specifies a condition for printing one or the other of two print lists. It works in the BREAK FOOTING, BREAK TITLE, DETAIL, PAGE FOOTING, PAGE TITLE, REPORT FOOTING, and REPORT TITLE commands.

You can nest IF/THEN/ELSE clauses.

```
IF cond-expr THEN (print-list) [ ELSE (print-list) ]
```

cond-expr

is a conditional expression that determines whether the THEN clause or the ELSE clause contains the list to print.

The conditional expression has the same form as an SQL search condition, except that it cannot include subqueries. It can include any form of column name (for example, COL 3 > 5 or EMPNUM = 228). Numeric expressions within the

conditional expression cannot use the AVG, COUNT, MAX, MIN, or SUM functions, but can include column identifiers and report writer functions. (For more information, see [Search Conditions](#) on page S-5 or [Expressions](#) on page E-21.)

THEN (*print-list*)

specifies what to print if the condition is true.

ELSE (*print-list*)

specifies what to print if the condition is false. If you omit the ELSE clause, report writer prints blanks when the condition is false.

print-list

is a list of items to print and optional formats for the items. It is the same as *print-list* for the DETAIL command, except that the HEADING, NOHEAD, NAME, SKIP, PAGE, and NEED clauses are not allowed. For more information, see [DETAIL Command](#) on page D-47.

Considerations—IF/THEN/ELSE

- If you want a heading for an IF/THEN/ELSE print list, you must specify one with the HEADING clause.
- The space required to print the result of an IF/THEN/ELSE is the length of the longest of the two print lists. The report writer pads the shorter list with blanks to the length of the longer list.
- You cannot subtotal or total the values in an IF/THEN/ELSE column, because these columns contain character values. For a technique you can use to print conditional values in a column that can be totaled and subtotaled, see the *SQL/MP Report Writer Guide*.

Examples—IF/THEN/ELSE

- Use IF/THEN/ELSE to flag an invalid job code:

```
S> DETAIL EMPNUM, LAST_NAME, IF JOBCODE > 0 THEN (JOBCODE)
+> ELSE ( "****" );
```

- Use IF/THEN/ELSE to print a default value as blanks:

```
S> DETAIL ORDERNUM,
+> IF DELIV_DATE <> 0 THEN ( DELIV_DATE AS DATE * );
```

- Use IF/THEN/ELSE to convert values to text:

```
S> DETAIL CUSTNUM, CUSTNAME,
+> IF CREDIT = "A1" THEN ("EXCELLENT") ELSE
+> ( IF CREDIT = "B1" THEN ("GOOD") ELSE
+> ( IF CREDIT = "C1" THEN ("FAIR")))
+> HEADING "CREDIT RATING";
```

IN Predicate

IN is a predicate that determines if a value is equal to any of the values in a list or collection of values.

```
expression1 [ NOT ] IN { { subquery }
                        { ( expression-list ) } }

expression-list is:

    expression [ , expression ] ...
```

subquery

is a subquery that has a result table of one column. For more information, see [Subqueries](#) on page S-82.

expression1 or expression

is an expression. For more information, see [Expressions](#) on page E-21.

The data type (including the character set for a character data type) of *expression1* must be compatible with the data type of the value returned by the subquery or the results of all expressions in the list.

Rules for comparisons of string and numeric values are the same as for comparison predicates. For more information, see [Comparison Predicate](#) on page C-58.

The maximum number of expressions you can specify in *expression-list* is 410.

Considerations—IN

- The IN predicate is true if either of this is true:
 - The first expression is equal to any expression in the list or to a value selected by the subquery.
 - The subquery returns no values.

The NOT operator reverses the value obtained from evaluating a search condition. For example, if IN is true, NOT IN is false, and so on.
- The IN predicate evaluates to null if either of this is true:
 - *expression1* evaluates to null.
 - The predicate is not true for any value returned by the subquery and the subquery returns at least one null value

Examples—IN

- This example finds those items whose number is 39, 337, or 452:

```
EMPNUM IN (39, 337, 452)
```

- The example finds those items whose part number is not in the PARTLOC table:

```
PARTNUM NOT IN (SELECT PARTNUM
                FROM INVENT.PARTLOC)
```

INCLUDE SQLCA Directive

INCLUDE SQLCA is a host program directive that declares the SQL communication area (SQLCA) in a host program.

The SQLCA is a status-checking area for host programs. SQL clears and reinitializes the SQLCA before each SQL statement executes. After the statement executes, SQL stores information about the success or failure of the statement in the SQLCA, including error and warning codes and messages.

For more information about the contents and usage of the SQLCA, see the SQL/MP programming manual for your host language.

```
INCLUDE SQLCA
```

Consideration—INCLUDE SQLCA

By default, INCLUDE SQLCA declares a Version 2 SQLCA. To request a different version, use INCLUDE STRUCTURES before INCLUDE SQLCA.

Example—INCLUDE SQLCA

This directives declare a version 315 SQLCA in a program:

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 315;
EXEC SQL INCLUDE SQLCA;
```

INCLUDE SQLDA Directive

INCLUDE SQLDA is a dynamic SQL directive that declares an SQL descriptor area (SQLDA) and optional names and collation buffers in a host program that uses dynamic SQL.

SQL uses the SQLDA with the dynamic SQL statements DESCRIBE and FETCH (to pass information about output columns) and DESCRIBE INPUT and EXECUTE (to pass information about input parameters). SQL uses the names buffer and collation buffer—which you can declare separately or with INCLUDE SQLDA—to pass column names and collation information returned by DESCRIBE and to pass input parameter names returned by DESCRIBE INPUT.

For more information about the contents and usage of SQLDA, see the SQL/MP programming manual for your host language.

```
INCLUDE SQLDA ( sqlda-name [, sqlvar-count ]
  [ , names-buffer, name-length ]
  [ , { RELEASE1 | RELEASE2 } ]
  [ , CPRULES collation-buffer, collation-size ] )
```

sqlda-name

is a host identifier that is the name for the SQLDA.

sqlvar-count

is an integer that specifies the maximum number of input parameters (including indicator parameters) or output columns to be described in the SQLDA at one time. The default is 1.

(PREPARE returns the number of input or output parameters in a statement to the INPUT-NUM or OUTPUT-NUM fields of the SQLSA when it compiles the statement. You can use these values for the corresponding SQLDA *sqlvar-count*.)

names-buffer, *name-length*

declares a names buffer.

names-buffer is the host variable name that is the name for the names buffer.

name-length is the maximum number of bytes in the longest column or parameter name that is returned to the buffer.

{ RELEASE1 | RELEASE2 }

is an obsolete clause for specifying that the version of the SQLDA should be 1 (RELEASE1) or 2 (RELEASE2). You cannot use this clause if you also use INCLUDE STRUCTURES. NonStop SQL/MP will not support this clause in the future, so you should use INCLUDE STRUCTURES to specify the version instead.

CPRULES *collation-buffer*, *collation-size*

declares a collation buffer.

collation-buffer is the host variable name that is the name for the collation buffer.

collation-size is the maximum number of bytes in the largest collation that is returned to the buffer.

Consideration—INCLUDE SQLDA

By default, INCLUDE SQLDA declares a version 2 SQLDA. To request a different version, use INCLUDE STRUCTURES before INCLUDE SQLDA.

The CPRULES clause can be used with SQL/MP versions 300 or later.

Example—INCLUDE SQLDA

This directives declare a version 310 SQLDA in a program:

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 310;
EXEC SQL INCLUDE SQLDA;
```

INCLUDE SQLSA Directive

INCLUDE SQLSA is a host program directive that declares the SQL statistics area (SQLSA) in a host program.

The SQLSA is an area in which SQL returns statistics about the execution or preparation of SQL statements. SQL clears the SQLSA before executing each statement, then returns statistics after the execution of a DELETE, FETCH, INSERT, OPEN, PREPARE, SELECT, or UPDATE statement. For PREPARE, statistics include information about input parameters and output columns associated with the dynamic SQL statement that was prepared.

For more information about the contents and usage of the SQLSA, see the SQL/MP programming manual for your host language.

INCLUDE SQLSA

Consideration—INCLUDE SQLSA

By default, INCLUDE SQLSA declares a version 2 SQLSA. To request a different version, use INCLUDE STRUCTURES before INCLUDE SQLSA.

Example—INCLUDE SQLSA

This directives declare a version 310 SQLSA in a program:

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 310;
EXEC SQL INCLUDE SQLSA;
```


INCLUDE STRUCTURES Directive

INCLUDE STRUCTURES is a host program or dynamic SQL directive that specifies the version of the structures generated by the INCLUDE SQLCA, INCLUDE SQLDA, and INCLUDE SQLSA directives.

```
INCLUDE STRUCTURES { [ ALL ] VERSION version }
                  { { SQLCA VERSION version } }
                  { { SQLDA VERSION version } }
                  { { SQLSA VERSION
                    { version | VERSION CURRENT } } }
```

version

is an integer that identifies a version of SQL/MP (1, 2, 300, 310, 315, 320, 325, or 330).

```
[ ALL ] VERSION version
```

specifies that any subsequently generated SQLCA, SQLDA, and SQLSA structures are to have the format for the SQL/MP version identified by *version*.

```
{ | SQLCA VERSION version | }
{ | SQLDA VERSION version | }
{ | SQLSA VERSION version | VERSION CURRENT | }
```

specifies that any subsequently generated SQLCA, SQLDA, or SQLSA structures are to have the format for the NonStop SQL/MP version identified by *version*. For SQLSA, you can specify either a particular version (by #) or VERSION CURRENT.

VERSION CURRENT allows C programs to use the most current SQLSA structure that the current system supports.

When you specify the VERSION CURRENT option, you must also include this line at the beginning of the C program source module:

```
#include <cextdecs (SQLGETSYSTEMVERSION)>
```

This line includes the prototype of the SQLGETSYSTEMVERSION procedure in your C program. For information and examples of programs that use this option, see the *C/C++ Programmer's Guide*.

If you specify version 330 of SQLSA in INCLUDE STRUCTURES, your pointers are automatically expanded to accommodate the larger structure described under [Considerations—INCLUDE STRUCTURES](#) on page I-8.

Considerations—INCLUDE STRUCTURES

- INCLUDE STRUCTURES can appear anywhere in the host language compilation unit where declarations are allowed, but it must precede any INCLUDE SQLCA, INCLUDE SQLDA, or INCLUDE SQLSA directive.

You should specify INCLUDE STRUCTURES once in every SQL module that contains an INCLUDE SQLCA, INCLUDE SQLDA, or INCLUDE SQLSA directive. New programs should generally use the INCLUDE STRUCTURES ALL VERSION form of the directive and specify the version of NonStop SQL/MP for which the program is written.

(INCLUDE SQLCA, INCLUDE SQLDA, and INCLUDE SQLSA generate version 2 structures unless an INCLUDE STRUCTURES directive in the same module specifies otherwise.)

- Version 330 of SQLSA returns more statistics information than earlier versions. Additional information returned includes the total processor time used by all sort processes (SORTPROGs) and all Executor Server Processes (ESPs) in a query. The size of the SQLSA structure has increased to accommodate these additions. In version 330, all SQLSA 16 bit counters are 32 bit. All 32 bit INT (32) counters in earlier versions are 64 bit (FIXED).

These version considerations apply only to SQLSA.

- You cannot use INCLUDE STRUCTURES if you also use the RELEASE option in the SQL directive or in the INCLUDE SQLDA directive. The host language compiler returns an error if you do so. (The RELEASE option is an older option that will be obsolete in the future. Use INCLUDE STRUCTURES instead.)

Examples—INCLUDE STRUCTURES

- This directive specifies version 330 for all structures declared later in the compilation unit:

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 330;
```

- This directive specifies version 310 for all SQLCA structures declared later in the compilation unit, but version 315 for all SQLDA and SQLSA structures declared later in the compilation unit:

```
EXEC SQL INCLUDE STRUCTURES SQLCA VERSION 310
                                SQLDA VERSION 315
                                SQLSA VERSION 315;
```

Index Keys

An index is stored in a key-sequenced file. Each row in an index contains:

- A two-byte column called the “keytag” column
- The columns specified in the CREATE INDEX statement
- The primary key of the underlying table (the user-defined primary key, the SYSKEY, or combination of the clustering key and the SYSKEY)

For a unique index, the primary key of the index is composed of the first two of these items. The primary key of the index cannot exceed 255 bytes, but the entire row (including the primary key of the index) can contain up to 510 bytes.

For a nonentity index, the primary key of the index is composed of all three items. The primary key cannot exceed 255 bytes. Because the primary key includes all the columns in the table, each row is also limited to 255 bytes.

For varying-length character columns, the length referred to in these byte limits is the defined column length, not the stored length. (The stored length is the expanded length, which includes two extra bytes for storing the data length of the item.)

The keytag value must be unique among indexes for the table; you can specify it when you create the index with the CREATE INDEX statement, or you can allow the system to generate it for you. (System-generated keytags are sequential numbers, beginning with one. User-specified keytag values can be either two bytes of character data or a SMALLINT UNSIGNED value in the range 1 through 65535. The keytag value for the primary key is 0.)

There is always a one-to-one correspondence between index rows and base table rows.

You should typically use random access to access index rows. Sequential access is less efficient for large subsets of rows unless SQL can use index-only access. For more information, see the *SQL/MP Query Guide*.

INDEXES Table

The INDEXES table is a catalog table that describes primary keys and indexes. [Table I-1](#) describes the contents of the INDEXES table.

Table I-1. The INDEXES Table

Column Name	Data Type	Description
1 TABLENAME *	CHAR(34)	Name of indexed table
2 INDEXNAME *	CHAR(34)	Name of index (name of primary key index is same as table name)
3 TABLECODE	SMALLINT UNSIGNED	Code for type of table; codes 100-999 mean reserved for HP use, other numbers are values of TABLECODE file attribute
4 COLCOUNT	SMALLINT SIGNED	Number of columns in index (includes keytag and, if table not unique, includes primary key columns)
5 CREATETIME	LARGEINT SIGNED	Julian timestamp from index creation
6 KEYTAG	SMALLINT UNSIGNED	Keytag specifier; 0 for primary key index
7 UNIQUEVALUE	CHAR(1)	Y if index is unique N if not
8 VALIDDEF	CHAR(1)	Y if index definition is valid (catalog tables and disk label are correct and consistent) N if not
9 VALIDDATA	CHAR(1)	Y if index has valid data N if not
10 INDEXLEVELS	SMALLINT SIGNED	Number of levels of indexing (maintained by UPDATE STATISTICS)
11 ROWSIZE	SMALLINT SIGNED	Length of packed index record
12 FILENAME	CHAR(34)	Name of file that contains index
13 SECURITYVECTOR	CHAR(4)	Guardian security string for the index
14 SECURITYMODE	CHAR(1)	Type of security in use: S Safeguard G Guardian
15 OBJECTVERSION	SMALLINT UNSIGNED	Version number of index

* Indicates primary key

The columns TABLENAME through FILENAME (1 through 12) were created in version 1. The columns SECURITYVECTOR through OBJECTVERSION (13 through 15) were added in version 300.

Guardian names in the INDEXES table are fully qualified and use uppercase characters. The Guardian security vector (column 13) is stored as uppercase characters.

Indicator Variables and Indicator Parameters

In a host program, a variable called an *indicator variable* is associated with each SQL data item that can contain a null value. The value of the indicator variable tells whether the corresponding data item is null (indicator is less than 0) or contains an actual value (indicator is 0).

Each indicator variable is a two-byte integer variable declared in the program. If you use INVOKE to generate record descriptions, INVOKE automatically includes an indicator variable for each item in the record that allows null values.

The INSERT, UPDATE, and SELECT statements use indicator variables. To send a null value to SQL for insertion, update, or comparison, you assign a value less than 0 to the indicator variable. To return a null value to your program, SQL sets the appropriate indicator variable to -1; to return a nonnull value, SQL sets the indicator variable to 0. (For INSERT or UPDATE, you can use the keyword NULL instead of an indicator variable.)

Indicator parameters serve the same purpose as indicator variables, but you use them to specify null input parameters in dynamic SQL or SQLCI statements. An indicator parameter appears following the associated input variable but separated by the key word INDICATOR. For example, this statement pass a negative indicator parameter (I) to SQL to indicate that the parameter P contains a null value:

```
SET PARAM ?I -1
INSERT INTO =EMPLOYEE VALUES (1000, ?P INDICATOR ?I);
```

For more information about using indicator variables or indicator parameters in host programs, see the SQL/MP programming manual for your host language.

INFO DEFINE Command

INFO DEFINE is an SQLCI command that displays the attributes and values associated with one or more existing DEFINES. (INFO DEFINE is similar to the TACL command INFO DEFINE and the OSS command info_define.)

```
INFO DEFINE { ( define [ , define ] ... ) } [ , DETAIL ] ;
```

define

is the name of an existing DEFINE or DEFINES for which you want information.

You can specify *define* as a DEFINE template. A DEFINE template allows you to use these special characters as part of a name:

** Matches all DEFINE names

=* Matches all DEFINE names

In TACL or OSS, but not in SQLCI, you can also use these special characters:

* Matches 0 or more characters at the same position

? Matches one character in the same position

DETAIL

requests the value of each attribute for the DEFINE that has a value. If you omit DETAIL, INFO DEFINE displays only the CLASS attribute and one other attribute (depending on CLASS).

Consideration—INFO DEFINE

INFO DEFINE does not display the working attribute set. (Use SHOW DEFINE to display the working attribute set.)

Examples—INFO DEFINE

- This example displays information about a DEFINE named =ORDERS:

```
>> INFO DEFINE =ORDERS;
DEFINE NAME      =ORDERS
CLASS            MAP
FILE             $VOL1.SALES.ORDERS
```

- This example displays information about the current =_DEFAULTS DEFINE:

```
>> INFO DEFINE =_DEFAULTS;
DEFINE NAME      =_DEFAULTS
CLASS            DEFAULTS
VOLUME          $VOL1.SALES
```

INITIALIZE SQL Command

INITIALIZE SQL is an SQLCI command that allows a user with the local super ID to ensure that SQL is using compatible components and to prepare a node to run NonStop SQL/MP. The INITIALIZE SQL command is required whenever you install a new NonStop SQL/MP PVU or software product revision (SPR).

```
INITIALIZE SQL ;
```

Considerations—INITIALIZE SQL

- If NonStop SQL/MP has not been installed on your node previously, you must create the system catalog before executing INITIALIZE SQL. (For more information, see [CREATE SYSTEM CATALOG Command](#) on page C-152.)

For more information on preparing your node to run NonStop SQL/MP, see the *SQL/MP Installation and Management Guide*.

- INITIALIZE SQL purges any existing SQLCI2 file, renames the file ZZSQLCI2 to SQLCI2, SQL-compiles SQLCI2, and registers the SQLCI2 program in the system catalog by executing these commands:

```
PURGE $SYSTEM.SYSTEM.SQLCI2
RENAME $SYSTEM.SYSTEM.ZZSQLCI2, $SYSTEM.SYSTEM.SQLCI2
SQLCOMP /IN $SYSTEM.SYSTEM.SQLCI2/ CATALOG $SYSTEM.SQL
```

Example—INITIALIZE SQL

To prepare NonStop SQL/MP on your node for the first time, enter these commands:

```
>> CREATE SYSTEM CATALOG;
>> INITIALIZE SQL;
```

INSERT Statement

INSERT is a DML statement that inserts a row into a table or protection view.

INSERT INTO { <i>name</i> } [(<i>column-list</i>)]	
{ VALUES (<i>val</i> [, <i>val</i>] ...) [<i>insert-opt</i>] ... } { (<i>select-stmt</i>) [<i>insert-opt</i>] ... } { <i>select-stmt</i> }	
<i>column-list</i> is:	[* [, <i>syskey</i>]] [<i>syskey</i> , *] [<i>col</i> [, <i>col</i>] ...]
<i>insert-opt</i> is:	{ [FOR] { STABLE } ACCESS } { { REPEATABLE } } { RETURNING { : <i>host-var</i> } } { { LASTSYSKEY } } { { ? <i>param</i> } } { { APPEND ANYWHERE } }

name

is the name of a table or protection view (or an equivalent DEFINE) in which to insert rows. *name* cannot be the name of a catalog table.

```
[ * [ , syskey ] ]
[ syskey , * ]
[ col [ , col ] ... ]
```

specifies the columns in the table or view in which to insert values (including null values) in the same order in which the values appear later in the statement:

*** All columns except the SYSKEY column

syskey The name of the SYSKEY column (usually SYSKEY)

col The unqualified name of a column

The default is all columns, except the SYSKEY column, in the order in which INVOKE would list them.

You can specify a SYSKEY column only for a table with relative file organization or for a view defined on such a table.

```
{ VALUES ( val [, val ] ...) [ insert-opt ] ... }
{ ( select-stmt ) [ insert-opt ] ... }
{ select-stmt }
```

specifies the values to insert. Include a value for each column specified previously on the INSERT statement, specify the values in the same order as

the columns, and specify values of appropriate type and size for the corresponding columns.

val

is a host variable, a literal, an expression, a parameter name, or the keyword NULL (representing a null value) that specifies a value. *val* cannot include a column reference.

select-stmt

specifies a select operation that selects values from other tables or views to insert in *name*. It has the syntax of a SELECT statement (see [SELECT Statement](#) on page S-18) with these restrictions:

- The select list must contain an element for each column specified on the INSERT statement.
- The SELECT cannot include a subquery that refers to a table, view, or underlying table of the view into which rows are being inserted.
- The SELECT cannot use the INTO clause.
- You must enclose the *select-stmt* in parentheses if it includes an access mode. The access mode applies only to rows compared to the selection criteria (in this case, even if the rows are in an audited table).

If *select-stmt* returns no rows, no rows are inserted. (Note that this contrasts with the behavior for subqueries in comparison predicates. For subqueries, if no values are returned, SQL returns a null value.)

[FOR] { STABLE | REPEATABLE } ACCESS

specifies STABLE or REPEATABLE access mode. STABLE is the default. (For more information, see [Access Options](#) on page A-1.)

RETURNING { *:host-var* }
 { LASTSYSKEY }
 { *?param* }

directs SQL to return the value of the SYSKEY for the last record inserted. (Applies only in host programs that INSERT into tables or views with a SYSKEY column.)

For static SQL programs, *:host-var* specifies a host variable to receive the SYSKEY. For dynamic SQL programs, LASTSYSKEY specifies a place holder to store the SYSKEY.

?param is the name of a parameter that has the same function as LASTSYSKEY and is included to maintain compatibility with existing programs. Use LASTSYSKEY instead.

APPEND | ANYWHERE

specifies whether to add rows at the end of the table (APPEND) or anywhere in the table (ANYWHERE). (Applies only to tables with relative file organization or to protection views defined on such tables. Cannot be used if the SYSKEY column is one of the columns for the INSERT.)

With APPEND, if you specify an ORDER clause in *select-stmt*, rows are added in that order. With ANYWHERE, an ORDER clause in *select-stmt* has no effect.

APPEND is the default unless the column list includes the SYSKEY.

Considerations—INSERT

- INSERT requires authority to read and write to the table or view receiving the data and authority to read tables and views specified in any *select-stmt* included in the INSERT statement.
- To insert a row, you must provide a value for each column in the table that has no default value. (As a result, you cannot insert a row into a protection view unless the view includes all columns of the underlying table that are defined with the NO DEFAULT option.)

In addition to being of appropriate type and size for the corresponding columns, the values in each row inserted must be compatible with the data types of the corresponding columns:

- Character values

Any character string data type is compatible with all other character string data types that have the same character set.

For character columns, inserted values shorter than the column length are padded on the right with single-byte ASCII blanks (HEX 20); longer values are truncated on the right. For varying-length character columns, shorter inserted values are not padded; values longer than the maximum length are truncated on the right.

- Numeric values

Any numeric data type is compatible with all other numeric data types.

If you insert a value into a numeric column that is not large enough, an overflow error occurs.

If a value has more digits to the right of the decimal point than specified by the scale for the column definition, the value is truncated.

- INTERVAL values

An INTERVAL data type is compatible only with another INTERVAL data type with the same range of INTERVAL fields.

- Date and time values

A date-time data type is compatible only with another date-time data type with the same range of DATETIME fields.

When you use a range of fields to specify only some of the DATETIME fields for a DATETIME column, SQL uses the current date and time for any missing fields to the left of the fields for which values are specified. For missing fields to the right of the fields for which values are specified, SQL uses these values:

YEAR	--	Current year	hour	--	00
MONTH	--	01	minute	--	00
DAY	--	01	second	--	00
			Fraction	--	000000

- SYSKEY values

For a table with relative organization, the value of a SYSKEY cannot exceed 4294963199 and cannot be greater than the maximum number of rows the table can contain. For a table with key-sequenced organization, the value of a SYSKEY cannot exceed 2^{63} minus 1.

If you insert rows into a protection view defined with a WHERE clause that refers to the SYSKEY column, you cannot specify APPEND or ANYWHERE and you must include SYSKEY in the column list. If such a protection view is also based on a key-sequenced or entry-sequenced table, you cannot insert a row into the view.

In addition, values in each row inserted must satisfy any constraints on the table or on the underlying table of the view. (A table constraint is satisfied if the check condition is not false - that is, it is either true or has an unknown value.) If the view is defined with WITH CHECK OPTION, the row must satisfy the view selection criteria specified in the WHERE clause of the AS *select-stmt* clause in the CREATE VIEW statement.

If a row does not qualify, SQL stops inserting rows and returns an error message.

- Use the CAST function to change the data types for input values specified in an INSERT statement. If a SELECT operation is specified in an INSERT statement, rather than input values, the CAST function must be used on the SELECT query output columns. For more information, see [CAST Function](#) on page C-4.
- To insert a null value, use the keyword NULL. From a program, you can also use an indicator variable to insert a null value, as described in the SQL/MP programming manual for your host language.
- To allow VSBB for insert operations for a nonaudited file, use the CONTROL TABLE directive with SEQUENTIAL INSERT ON, SYNCDEPTH 0, and TABLE LOCK ON options. Additionally, specify FOR REPEATABLE ACCESS in your INSERT statement and specify IN EXCLUSIVE MODE in the LOCK TABLE statement.

For information on buffering INSERT operations, see [CONTROL TABLE Directive](#) on page C-77.

- In host programs, status for INSERT operations is reported to the SQLCODE variable in the SQLCA:

< 0	An error code number
0	INSERT was successful
> 0	A warning code number
100	No rows qualify for an INSERT through a SELECT

Some error/warning codes have both a positive and a negative version because the problem described by the associated message causes an error in some situations and a warning in others.

The SQLCA also records the number of rows inserted.

Examples—INSERT

- This example inserts a row into the CUSTOMER table and supplies the value "A2" for the CREDIT column:

```
INSERT INTO SALES.CUSTOMER ( * )
VALUES ( 4777, "ZYROTECHNIKS", "11211 40TH ST.",
        "BURLINGTON", "MASS.", "01803", "A2" );
```

- This example also inserts a row into the CUSTOMER table. Unlike the previous example, this INSERT does not include a value for the CREDIT column, which has a default value. As a result, this INSERT must include the column name list.

```
INSERT INTO SALES.CUSTOMER
(CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE)
VALUES (1120, "EXPERT MAILERS", "5769 N. 25TH PLACE",
        "PHOENIX", "ARIZONA", "85016");
```

- This example inserts a DATETIME value into the TIME_SHIPPED column:

```
INSERT INTO SHIPMENTS (TIME_SHIPPED)
VALUES ( DATETIME "1988-10-22:08:15" YEAR TO MINUTE );
```

- This example inserts a DATE value into the BIRTHDATE column of the PERSONNEL table:

```
INSERT INTO PERSONNEL (BIRTHDATE)
VALUES ( DATE "1940-10-09" );
```

- This example inserts DATETIME and INTERVAL values:

```
INSERT INTO PROJECTS
VALUES ( "945", DATETIME "1989-10-20" YEAR TO DAY,
        DATETIME "1990-10-21" YEAR TO DAY,
        INTERVAL "30" DAY );
```

- In this example, CUSTLIST is a protection view of all columns of the CUSTOMER table except the credit rating.

Suppose that one of your suppliers has become a customer. If you can use the same number for both the customer and supplier numbers, you can select the supplier information from the SUPPLIER table and insert it in the CUSTOMER table through the CUSTLIST view. This operation works because the columns of the SUPPLIER table contain values that correspond to the columns of the CUSTLIST view. If you want a credit rating that is different from the default, you must update the row.

```
VOLUME $VOL1.SALES;
INSERT INTO CUSTLIST
  ( SELECT * FROM INVENT.SUPPLIER WHERE SUPPNUM = 10 );
UPDATE CUSTOMER SET CREDIT = "A4" WHERE CUSTNUM = 10;
```

INTERVAL Data Type

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

No INTERVAL unit exists to bridge a year-month interval or a day-time interval because the varying number of days in a month makes conversion on a duration basis inexact.

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

start-ym and *end-ym* are:

```
{ YEAR }
{ MONTH }
```

but the *start-ym* you specify must precede the *end-ym* you specify in the list.

start-dt and *end-dt* are:

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

but the *start-dt* you specify must precede the *end-dt* you specify in the list, and only *end-dt* can include the *precision* option for FRACTION.

start-ym [(*digits*)] [TO *end-ym*]

specifies the range of fields for a year-month set of INTERVAL values, or a subset of the year-month INTERVAL values, and the number of digits allowed for the starting field in the set.

start-dt [(*digits*)] [TO *end-dt*]

specifies the range of fields for a day-time set of INTERVAL values, or a subset of the day-time INTERVAL values, and the number of digits allowed for the starting field in the set.

digits

is an unsigned integer from 1 to 18 that specifies the number of significant digits for the first field in the set. For example, YEAR(2) allows up to 99 years; YEAR(4) allows up to 9999 years.

The default is 2 digits. The maximum number of digits in the starting field depends on the number and size of the remaining fields in the set; the entire INTERVAL value can contain no more than 18 digits.

precision

is an unsigned integer in the range 1 through 6 that specifies the number of significant digits with which to express the fraction of a second. The default is 6.

Considerations—INTERVAL Data Type

- A specific INTERVAL data type is compatible only with another INTERVAL data type that has the same range of INTERVAL fields.
- An INTERVAL value can have a maximum of 18 digits, including the digits in all fields.

Any INTERVAL field that is a starting field can have up to 18 digits minus the number of other digits in the INTERVAL value (but the starting field will have only 2 digits unless you specify a larger value with the *digits* option). The maximum value for the starting field is the maximum value that can be expressed in the number of digits allowed for the field.

If an INTERVAL field is not a starting field, the maximum value of digits in the field is:

YEAR	(Always a starting field)
MONTH	1 to 11
DAY	(Always a starting field)
hour	1 to 23

MINUTE	1 to 59
SECOND	1 to 59
FRACTION	1 to 999999 (less with small precision)

An INTERVAL can be negative, but individual fields within the interval are expressed as positive values. The negative sign (-), if present, applies to the entire value, and is not counted in the number of digits for any field.

- To compute the size of an INTERVAL column:
 - Add 1 byte for the sign.
 - For the starting field, add 2 bytes for 1 to 4 digits, 4 bytes for 5 to 8 digits, and 8 bytes for 9 to 18 digits.
 - Add 2 bytes for each nonstarting field other than FRACTION.
 - If a FRACTION field is present and is not a starting field, add 2 bytes for a precision of 1 to 4 significant digits and 4 bytes for a precision of 5 or 6 digits.
 - If the column allows null values, add 2 bytes.

You can also determine the storage size for a column by querying the COLSIZE column of the COLUMNS catalog table. For example, this query from SQLCI returns a column's length in bytes:

```
>>SELECT colsize FROM columns
+> WHERE tablename LIKE "%table-name%"
+> AND colname = "column-name";
* Version Management Consideration
```

The INTERVAL data type is supported on NonStop SQL/MP versions 2 and later.

Example—INTERVAL Data Type

This statement creates a table in which three of the four columns are of data type INTERVAL. Column AGE represents an interval of years and months (for example 27-2, which means 27 years and 2 months), column YRS_EXPERIENCE represents an interval of years, and column HOURS_VACATION represents an interval of hours.

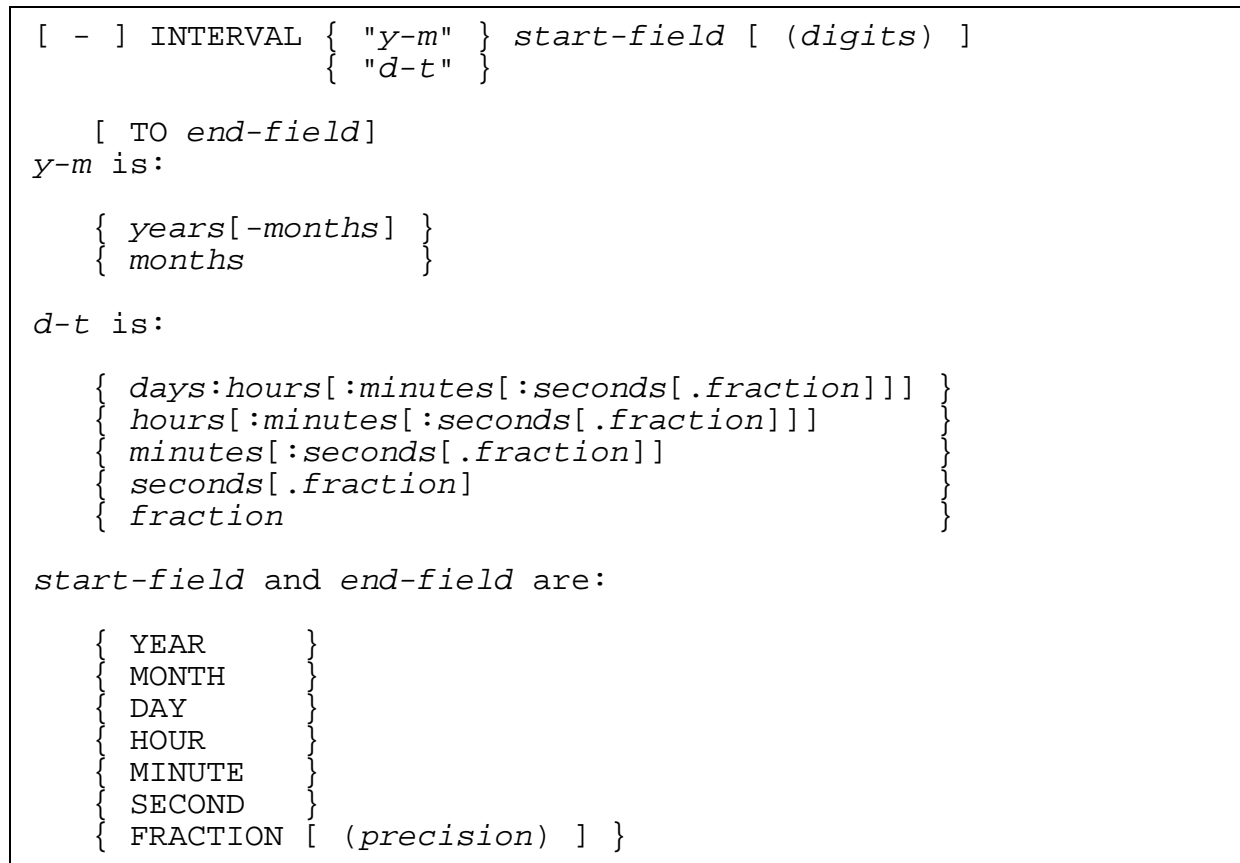
```
CREATE TABLE EMPLOYEE (
    AGE                INTERVAL YEAR TO MONTH,
    NAME               PIC X(30) NO DEFAULT NOT NULL,
    YRS_EXPERIENCE     INTERVAL YEAR,
    HOURS_VACATION     INTERVAL HOUR(3) NOT NULL
)
```

YRS_EXPERIENCE can be no more than 99 (the default is two digits), but HOURS_VACATION can be up to 999 because the CREATE TABLE statement explicitly specifies three digits.

INTERVAL Literals

An INTERVAL literal is a constant of data type INTERVAL that represents a positive or negative duration of time as a year-month or day-time interval.

An INTERVAL literal can contain a maximum of 18 digits, plus characters such as hyphens (-) or colons (:) that separate the values of INTERVAL fields. The value can be enclosed in either double quotation marks (shown in this diagram) or in single quotation marks.



The *start-field* you specify must precede the *end-field* you specify in the list of field name, and only *end-field* can use the *precision* option on FRACTION.

years

is an unsigned integer that specifies a number of years. It can have up to 18 digits, minus the number of digits in the *months* field, if any. Negative values are allowed, with the minus sign inside the quotes.

months

is an unsigned integer that specifies a number of months. Used as a starting field, it can have up to 18 digits; as a nonstarting field, it must be in the range 0 through 11. Negative values are allowed, with the minus sign inside the quotes.

days

is an unsigned integer that specifies a number of days. It can have up to 18 digits, minus the number of digits in the other fields of the INTERVAL literal. Negative values are allowed, with the minus sign inside the quotes.

hours

is an unsigned integer that specifies a number of hours. Used as a starting field, it can have up to 18 digits, minus the number of digits in the other fields of the INTERVAL literal; as a nonstarting field, it must be in the range 0 through 23. Negative values are allowed, with the minus sign inside the quotes.

minutes

is an unsigned integer that specifies a number of minutes. Used as a starting field, it can have up to 18 digits, minus the number of digits in the other fields of the INTERVAL literal; as a nonstarting field, it must be in the range 0 through 59. Negative values are allowed, with the minus sign inside the quotes.

seconds

is an unsigned integer that specifies a number of seconds. Used as a starting field, it can have up to 18 digits, minus the number of digits in the other fields of the INTERVAL literal; as a nonstarting field, it must be in the range 0 through 59. Negative values are allowed, with the minus sign inside the quotes.

fraction

is an unsigned integer that specifies a fraction of a second. Used as a starting field, it can have up to 18 digits; as an ending field, it is limited to the number of digits specified by *precision*.

start-field [(*digits*)] [TO *end-field*]

specifies the range of INTERVAL fields in the literal and the number of digits allowed in the starting field. The default for *digits* is 2. (For more information about INTERVAL fields, see [INTERVAL Data Type](#) on page I-19.)

precision

is an unsigned integer in the range 1 to 6 that specifies the number of significant digits in the portion of the literal that specifies the fraction of a second. The default is 6.

Example—Interval Literals

These are all INTERVAL literals:

INTERVAL "1" MONTH	An interval of 1 month
INTERVAL "7" DAY	An interval of 7 days
INTERVAL "2-7" YEAR TO MONTH	An interval of 2 years, 7 months
INTERVAL "5:2:15:36.8" DAY TO FRACTION(1)	An interval of 5 days, 2 hours, 15 minutes, and 36.8 seconds
- INTERVAL "5" DAY	An interval that subtracts 5 days

INVOKE Directive and Command

INVOKE is a directive or SQLCI utility command that produces a record description that corresponds to a row in a specified table or view.

The record description includes a data item for each column in the table or view except the SYSKEY column and (except for SQL-format) an indicator variable for each column that allows null values. The record description includes the SYSKEY column of a view only if the view explicitly listed the column in its definition. Because INVOKE declares host variables that are compatible with the SQL columns, no data conversion is required at run time.

Used in a host program Declare Section, INVOKE creates a host program record description directly in the program. Used from SQLCI, INVOKE writes the record description on the OUT file and, optionally, in an EDIT file.

INVOKE { <i>name</i> }	[AS <i>record</i>]
	[]
	[]
	[FORMAT {	}
	[[ANSI TANDEM] COBOL85	}
	[PASCAL	}
	[SQL	}
	[TAL	}
	[]
	[LEVEL {	}
	[<i>base</i>	}
	[(<i>base</i> , <i>inc</i>)	}
	[]
	[DATEFORMAT {	}
	[DEFAULT	}
	[EUROPEAN	}
	[USA	}
	[]
	[{ { PREFIX <i>indicator-prefix</i> }	}
	[{ { SUFFIX <i>indicator-suffix</i> }	}
	[]
	[{ NULL STRUCTURE	}
	[]
	[TO file [(<i>section</i>)] [CLEAR]]
	[]
	[CHAR AS { STRING ARRAY }]

{ *name* }

is the name of an existing table or view for which to create a record description. It can be a DEFINE name.

If SMF is installed on your node, *name* cannot be on any \$.ZYS*. subvolumes.

AS *record*

specifies the name for the record. *record* must be a host language identifier and cannot be a DEFINE name.

If you omit the AS clause, the record name depends on the FORMAT option. For COBOL85, the record name is the unqualified name of the table or view. (For example, if the table name is \SYS1.\$VOL1.PERSNL.JOB, the record name is JOB.) For C or PASCAL, the record name is the unqualified name of the table or view with the suffix “_type” appended. (For example, JOB_TYPE.) For TAL, the record name is the unqualified name of the table or view with the suffix “^type” appended and an asterisk indicating a structure template. (For example, JOB^TYPE(*).)

```
FORMAT { C | [ANSI|TANDEM] COBOL85 | PASCAL | SQL | TAL }
```

specifies the language format for the record definition.

The default depends on the environment. In a C program, the default is C; in a TAL program, the default is TAL; and so forth. In SQLCI, the default is SQL.

The COBOL85 format can be ANSI or TANDEM. In ANSI format, each fixed format line has a sequence number and contains a maximum of 80 characters; in TANDEM format, a free format line has no sequence number and contains a maximum of 132 characters. The default for COBOL85 is TANDEM COBOL85.

```
LEVEL { base  
      { ( base, inc ) }
```

(for use in COBOL programs or from SQLCI) specifies an integer in the range 1 to 49 as the base level number for a COBOL85 record definition and, optionally, an integer in the range 1 to 24 as the increment used to assign level numbers to data items (columns) in the record.

The base plus two times the increment must not exceed 49. If the increment is too large, SQL issues a warning and uses 1. The default is LEVEL 01, 01.

```
DATEFORMAT { DEFAULT | EUROPEAN | USA }
```

specifies the format of host variables for date-time columns.

For a column with a date-time data type that has an HOUR field, DATEFORMAT USA causes INVOKE to produce a host variable that is three bytes longer than an equivalent host variable for EUROPEAN or DEFAULT format. The extra bytes allow room for “am” or “pm” following the values.

For information on the DEFAULT, EUROPEAN, or USA formats, see [Date-Time Literals](#) on page D-10.

The default is DATEFORMAT DEFAULT.

```
{ | PREFIX indicator-prefix | }  
{ | SUFFIX indicator-suffix | }
```

specifies a prefix, a suffix, or both for indicator variable names. The names have the form:

indicator-prefix column-name indicator-suffix

If you do not specify a prefix, indicator variable names have no prefix. If you specify a prefix but do not specify a suffix, indicator names have no suffix. If you do not specify either a prefix or a suffix, the suffix depends on the language:

C	<u>i</u>	Pascal	<u>i</u>
COBOL	-I	TAL	^i

A prefix or suffix must consist of legal identifier values for the host language in which it is used. However, you can use uppercase or lowercase letters in a prefix or suffix, regardless of the host language. For C, Pascal, or TAL, INVOKE converts the suffix to lowercase; for COBOL, INVOKE converts the suffix to uppercase.

If you specify C, PASCAL, or TAL format and the indicator variable name with suffix is longer than 31 characters, the name is truncated to 31 characters. If you specify COBOL format and the indicator variable name with suffix is longer than 30 characters, the name is truncated to a length of 30 and any trailing separator characters (-) are removed. A warning is issued for each truncated name.

For more information about indicator variables, see [Indicator Variables and Indicator Parameters](#) on page I-11 or the SQL programming manual for your host language.

NULL STRUCTURE

specifies that a column that allows null values should be declared as a structure with the same name as the column and with fields for the data item and its indicator variable. The fields are named INDICATOR (or indicator) and VALU (or value).

TO *file* [(*section*)] [CLEAR]

(for use from SQLCI) specifies an EDIT file on which to write the record definition. *file* can be a DEFINE name. If *file* does not exist, INVOKE creates it. If SMF is installed on your node, *file* must be either a logical or direct file.

section is a host identifier that names a section in the file. If you specify *section*, INVOKE writes the directive ?SECTION *section* immediately before the record description in the EDIT file.

CLEAR directs INVOKE to purge any data in the file before writing the record description. If you omit CLEAR, INVOKE adds the record description after the existing data.

CHAR AS { STRING | ARRAY }

(for use from C programs or from SQLCI) specifies whether to create a byte for the null terminator in C character types:

STRING	Generate the extra byte
ARRAY	Omit the extra byte

Considerations—INVOKE

- To use the INVOKE statement on a table or view, you must have authority to read the table or view at compile time.
- Multibyte characters can be displayed only on output devices that support them. If a record definition contains a DEFAULT clause with a multibyte character, the output might not be displayed on your output device.

Examples—INVOKE

- This SQLCI command generates an SQL-format description of a table named EMPLOYEE:

```
>> INVOKE $VOL1.PERSNL.EMPLOYEE FORMAT SQL;
-- Definition of table \SYS1.$VOL1.PERSNL.EMPLOYEE
-- Definition current at 10:43:27 - 03/01/94
(
  EMPNUM                NUMERIC( 4, 0 ) UNSIGNED NOT NULL
, FIRST_NAME            CHAR( 15 ) NOT NULL
, LAST_NAME             CHAR ( 20 ) NOT NULL
, DEPTNUM               NUMERIC( 4, 0 ) UNSIGNED NOT NULL
, JOBCODE               NUMERIC( 4, 0 ) UNSIGNED
, SALARY                NUMERIC( 8, 2 ) UNSIGNED
)
```

- The next SQLCI command generates a COBOL-format record named EMP that corresponds to a table named EMPLOYEE. SQLCI appends the record description to an EDIT file name COBLIB in a section named EMPSEC and displays the record description on the OUT file as well.

Compare the record description in the example to the SQL-format record description in the last example and notice the indicator variables for JOBCODE-I and SALARY-I that are included in the COBOL format but not in the SQL format.

```
>> INVOKE $VOL1.PERSNL.EMPLOYEE AS EMP FORMAT COBOL
+>   TO COBLIB (EMPSEC);
?SECTION EMPSEC
* Definition of table \SYS1.$VOL1.PERSNL.EMPLOYEE
* Definition current at 10:50:32 - 03/01/94
01 EMP.
  02 EMPNUM                PIC 9(4) COMP.
  02 FIRST_NAME            PIC X(15)
  02 LAST_NAME             PIC X(20)
  02 DEPTNUM               PIC 9(4) COMP.
  02 JOBCODE-I             PIC S9(4) COMP.
  02 JOBCODE               PIC 9(4) COMP.
  02 SALARY-I              PIC S9(4) COMP.
  02 SALARY                PIC 9(6)V9(2) COMP.
```

ISLACK File Attribute

ISLACK is a file attribute that specifies the minimum percentage of space to leave for future insertions when loading index blocks. ISLACK applies only to key-sequenced tables and to indexes.

ISLACK <i>percent</i>

The default is the value of the SLACK file attribute.

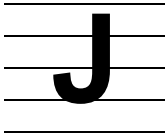
The default for SLACK is 15 percent.

percent

is an integer from 0 to 99 that specifies the percent of empty space to leave in each index block during loading.

Considerations—ISLACK

- ISLACK specifications are usually between 15 and 25 percent.
- Specifying a larger-than-normal ISLACK value when a file is initially loaded, and many more inserts are expected, can improve performance by reducing the number of block splits required when inserts occur.
- For a file expected to have little activity, you can save disk space by specifying a smaller-than-normal ISLACK value.



Joins

A join is an operation that combines two tables or views to form a new table. A join query is a query that requests columns from more than one table or view.

A join query should contain predicates that compare a column from one table with a column from another table. The join concatenates (joins together) rows (from each of the joined tables) that satisfy the predicates. Without predicates, SQL creates a Cartesian product with all rows of each table combined with each other.

NonStop SQL/MP supports two types of joins: inner joins and left outer joins.

An inner join discards rows that do not satisfy the predicates specified in a WHERE clause or an ON clause. Inner joins are useful for reporting information that satisfies a given set of requirements.

An outer join returns all rows from one or more of the tables being joined; rows that do not satisfy the search condition have missing information in columns that correspond to the other tables being joined. An outer join is useful for generating exception reports (retrieving information that does NOT satisfy a stated set of requirements).

A left outer join returns all rows from the left table or view - the table or view left of the keywords LEFT JOIN in the SELECT statement - and rows from the other table that satisfy the search condition. A left outer join is not necessarily symmetric; A LEFT JOIN B is not necessarily the same as B LEFT JOIN A.

NonStop SQL/MP allows you to specify the type of algorithm used for a join operation and to specify the sequence of joins within a SELECT statement. For information, see [CONTROL QUERY Directive](#) on page C-74 (the HASH JOIN option) and [CONTROL TABLE Directive](#) on page C-77 (the JOIN METHOD and JOIN SEQUENCE options).

For information on joins, see the *SQL/MP Query Guide*.

Examples - Joins

These examples refer to a database consisting of:

EMPLOYEE TABLE

EMP_ID	LAST_NAME	FIRST_NAME	DEPT_NUM	MGR_ID	SALARY
2703	Smith	James	7620	2705	47500.00
2705	Simpson	Travis	7600	6554	68000.00
2906	Nakagawa	Etsuro	6400	6554	72000.00
3598	Nakamura	Eichiro	6480	2906	50000.00
4096	Murakami	Kazuo	6410	3598	36000.00
5361	Smythe	Roger	7690	9069	42650.00
9069	Smith	John	7690	2705	38760.00

DEPT TABLE

DEPT_NUM	DEPT_NAME	DEPT_LOC
6400	Marketing - Far East	900
6410	Marketing - Korea	910
6420	Marketing - Hong Kong	920
6440	Marketing - Singapore	940
6470	Marketing - Taiwan	970
6480	Marketing - Australia	980
7600	Marketing - USA	100
7620	Marketing - USA West	120

- Examples of inner joins

In these examples, rows that do not have the same department number are not returned in the result. So, rows with employee number 5361 and 9069 do not appear in the result because the corresponding department number value does not exist in the DEPT table.

This query uses an inner join of the tables EMPLOYEE and DEPT:

```
SELECT E.LAST_NAME, E.FIRST_NAME, E.DEPT_NUM,
       D.DEPT_NUM, D.DEPT_NAME
FROM   EMPLOYEE E, DEPT D
WHERE  E.DEPT_NUM = D.DEPT_NUM
```

This example shows an equivalent query that uses an INNER JOIN operator:

```
SELECT E.LAST_NAME, E.FIRST_NAME, E.DEPT_NUM,
       D.DEPT_NUM, D.DEPT_NAME
FROM   EMPLOYEE E INNER JOIN DEPT D
ON     E.DEPT_NUM = D.DEPT_NUM
```

Both queries return this result:

Murakami	Kazuo	6410	6410	Marketing - Korea
Nakagawa	Etsuro	6400	6400	Marketing - Far East
Nakamura	Eichiro	6480	6480	Marketing - Australia
Simpson	Travis	7600	7600	Marketing - USA
Smith	James	7620	7620	Marketing - USA West

- Examples of left outer joins

This query retrieves all rows from the EMPLOYEE table. Employees with employee numbers 5361 and 9069 are in department 7690. Because there is no matching department number value in the DEPT table, columns selected from the

DEPT table contain a question mark (?) to denote missing information. Each question mark represents a null value.

```
SELECT E.LAST_NAME, E.FIRST_NAME, E.DEPT_NUM,
       D.DEPT_NUM, D.DEPT_NAME
FROM EMPLOYEE E LEFT JOIN DEPT D
      ON E.DEPT_NUM = D.DEPT_NUM
```

The query returns this data:

Murakami	Kazuo	6410	6410	Marketing - Korea
Nakagawa	Etsuro	6400	6400	Marketing - Far East
Nakamura	Eichiro	6480	6480	Marketing - Australia
Simpson	Travis	7600	7600	Marketing - USA
Smith	James	7620	7620	Marketing - USA West
Smythe	Roger	7690	?	?
Smith	John	7690	?	?

In an outer join, the WHERE clause restricts the result. For example, to get rows related only to employees who are the exceptions, check for a null value in the DEPT_NUM column of the DEPT table:

```
SELECT E.LAST_NAME, E.FIRST_NAME, E.DEPT_NUM,
       D.DEPT_NUM, D.DEPT_NAME
FROM EMPLOYEE E LEFT JOIN DEPT D
      ON E.DEPT_NUM = D.DEPT_NUM
WHERE D.DEPT_NUM IS NULL
```

The query returns this data:

Smythe	Roger	7690	?	?
Smith	John	7690	?	?

The IS NULL predicate is applied to the DEPT_NUM column of the DEPT table because it appears in the join predicate and belongs to the table that is not preserved.

A null value marker (?) in the column indicates that for a given department number in the EMPLOYEE table, there is no matching department number in the DEPT table. You can refine the report by eliminating columns selected from the DEPT table:

```
SELECT E.LAST_NAME, E.FIRST_NAME, E.DEPT_NUM
FROM EMPLOYEE E LEFT JOIN DEPT D
      ON E.DEPT_NUM = D.DEPT_NUM
WHERE D.DEPT_NUM IS NULL
```

The query returns this data:

Smythe	Roger	7690
Smith	John	7690

JULIANTIMESTAMP Function

JULIANTIMESTAMP is a function that converts a date-time value into a 64-bit Julian timestamp that represents the number of microseconds that have elapsed between 4713 B.C., January 1, 00:00 and the specified date-time value.

JULIANTIMESTAMP returns a value of type LARGEINT. You can use JULIANTIMESTAMP anywhere SQL allows a numeric expression.

<code>JULIANTIMESTAMP (<i>date-time-expression</i>)</code>
--

date-time-expression

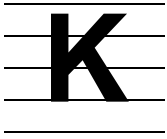
is an expression that evaluates to a value of type DATETIME, DATE, TIME, or TIMESTAMP.

If *date-time-expression* does not contain all the fields from YEAR through FRACTION, SQL extends the value (using the EXTEND function rules) before converting it to a Julian timestamp.

Example—JULIANTIMESTAMP

This example converts a date-time value into a Julian-timestamp representation of the value. (If START_DATE is 1988-02-21:20:30, the resulting Julian timestamp would be 211439233800000000.)

```
SELECT JULIANTIMESTAMP ( START_DATE ) FROM PROJECTS
WHERE PROJECT_NAME = "920";
```



Keys

See one of these specific entries:

[Clustering Keys](#)

[Index Keys](#)

[Partitions](#) (first keys)

[Primary Keys](#)

[Syskeys](#) (system-defined primary keys)

[User-Defined Keys](#) (user-defined primary keys)

KEYS Table

The KEYS table is a catalog table that describes the columns of primary keys and indexes. [Table K-1](#) describes the contents of the KEYS table.

Table K-1. The KEYS Table

Column Name	Data Type	Description
1 INDEXNAME *	CHAR (34)	Name of index
2 KEYSEQNUMBER *	SMALLINT UNSIGNED	Number indicating position of column in index row
3 TABLECOLNUMBER	SMALLINT UNSIGNED	Number indicating position of column in table row
4 ORDERING	CHAR (1)	A if ascending order D if descending order
5 CPRULESNAME	CHAR (34)	Name of the collation associated with the primary key or index

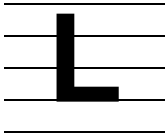
* Indicates primary key

The columns INDEXNAME through ORDERING (1 through 4) were created in version 1. Column 5, CPRULESNAME, was added in version 300.

The KEYS table describes the columns of the primary key index and the columns of indexes defined with CREATE INDEX. The primary key is handled as an index with the same name as the table.

The first entry for an index defined with CREATE INDEX is KEYTAG, although KEYTAG is not a column of the table being indexed. For the KEYTAG column, the TABLECOLNUMBER column has the value 65535.

Guardian names in KEYS are fully qualified and use uppercase characters.



LEFT_MARGIN Option

LEFT_MARGIN is an option of the SQLCI report writer SET LAYOUT command that sets the left margin for the current report and for subsequent reports until you reset it.

LEFT_MARGIN *number*

number

specifies the number of spaces to precede the leftmost printed character in the report. *number* must be an integer in the range 0 through 255 and cannot exceed the value of the RIGHT_MARGIN layout option. The default is zero.

Each space corresponds to a position that could be occupied by one single-byte character, regardless of the character set in use.

Example—LEFT_MARGIN

This example sets the left margin to 10:

```
>> SELECT * FROM PERSNL.JOB;
S> LIST FIRST 1;
JOBCODE  JOBDESC
-----
      100  MANAGER

S> SET LEFT_MARGIN 10;
S> LIST FIRST 1;
          JOBCODE  JOBDESC
          -----
          100  MANAGER
```

LIKE Predicate

LIKE is a predicate that searches for character strings matching a pattern.

```
char-exp [NOT] LIKE pattern [ ESCAPE char ]
      [ TERMINATE { char } ]
```

char-exp

is a character expression that specifies the set of strings to search for matches to *pattern*.

pattern

is a character expression that does not contain a column name and that specifies the pattern string for the search.

ESCAPE *char*

specifies a literal, host variable (preceded by a colon), or parameter that contains a single character to use as an escape character to turn off the special meaning of percent and underscore.

If the column is associated with a single-byte character set, *char* must be one single-byte character. If the column is associated with a double-byte character set, *char* must be one double-byte character.

TERMINATE *char*

specifies a literal, host variable (preceded by a colon), or parameter that contains a single character to use to indicate the end of the pattern within the pattern string. Use this clause when the column value and the comparison value are different lengths.

If you specify both ESCAPE and TERMINATE, the values for *char* must be different in each clause.

If the column is associated with a single-byte character set, *char* must be one single-byte character. If the column is associated with a double-byte character set, *char* must be one double-byte character.

Considerations—LIKE

- The values you compare must be character strings. Lowercase and uppercase letters are not equivalent. To make lowercase letters match uppercase letters, use UPSHIFT.
- The LIKE predicate is true if the value matches any string in the column to which you compare the value.

- If the values you compare are both empty strings (that is, strings of zero length), the LIKE predicate is true.
- A blank is compared in the same way as any other character.
- If the value of *col-name* is null or if *host-variable* or *param-name* contains INDICATOR clauses that specify a null value, the LIKE predicate evaluates to null.
- You can use only parameters in a prepared dynamic SQL statement or in a statement you enter through SQLCLI.
- If you specify NOT, the predicate is true if the value you are comparing does not match any string to which you compare it, or is not the same length as any string to which you compare it. For example, NAME NOT LIKE “_Z” is true if the string is not two characters long or the last character is not Z.

In a *search-condition*, the predicate NAME NOT LIKE “ABC” is equivalent to NOT (NAME LIKE “ABC”).

- You can look for similar values by specifying only a few characters and using these wild-card characters:
 - % Percent sign indicates zero or more characters of any type are acceptable. For example, “%ART%” matches “SMART”, “ARTIFICIAL”, and “PARTICULAR”, but not “smart”.
 - _ Underscore indicates any single character is acceptable. For example, “BOO_” matches “BOOK” or “BOOR”; but not “BOO”, “BOOKLET”, or “book.”

You must specify the wild-card characters (underscore and percent sign) in the character set associated with the column or unexpected results can occur. These are the values for the character sets SQL supports:

Character Set	Underscore	Percent Sign
UNKNOWN	HEX 5F	HEX 25
ISO99591/9	HEX 5F	HEX 25
KANJI	HEX8151	HEX 8193
KSC5601	HEX A3DF	HEX A3A5

- If you want to search for a string containing a percent sign or underscore, you can define an escape character (using `ESCAPE character`) to turn off the special meaning of percent sign and underscore.

You can specify a character as a host variable, string literal, or a parameter name for which you supply a one-character value later (for example, ?ESC). If you want to include a percent sign or underscore in the comparison string, enter the escape character immediately preceding it. For example, to locate the value A_B, enter:

```
NAME LIKE "A\_B" ESCAPE "\"
```

To include the escape character itself in the comparison string, enter two escape characters. For example, to locate A_B\C%, enter:

```
NAME LIKE "A\_B\\C\%" ESCAPE "\"
```

The escape character must precede only the percent sign, underscore, or escape character itself. For example, if the escape character is \, RA\BS is not valid.

- These guidelines apply to pattern matching using columns of data type CHARACTER:
 - Columns of data type CHARACTER are fixed length.

When a value is entered, SQL pads the value in the column with blanks if necessary. The value “JOE” inserted in a CHAR(6) column becomes “JOE ” (3 characters plus 3 blanks).

In a comparison value, the condition is met only if the column value and the comparison value are the same length. The value “JOE ” does not match “JOE” but matches “JOE%”.
 - The TERMINATE clause is useful in host programs where the pattern appears in a host variable of data type CHARACTER. This example finds all names that contain a *y*:

```
MOVE "%y%@" TO hostvar
```

```
WHERE NAME LIKE :hostvar TERMINATE "@"
```

- These guidelines apply to pattern matching using columns of varying-length character data types:
 - Columns of varying-length character data types do not include trailing blanks unless blanks are specified when data is entered. For example, the value “JOE” inserted in a VARCHAR(4) column is “JOE”. The value matches both “JOE” and “JOE%.”
 - If you cannot locate a value in a varying-length character column, it might be because trailing blanks were specified when the value was inserted into the table. For example, a value of “5MB__” is not located by LIKE “%MB,” but by “%MB%.”
 - The TERMINATE clause specifies the end of a pattern within a pattern-matching string. For example, column NAME, defined as a VARCHAR column, contains these values in the EMPLOYEE table:

```
NAME
----
Jay
Mike
Holly
Dave
```

Suppose that you want to select all names that end with the letter *y*. If you define the pattern-matching string as “%*y*@,” this statement finds all names ending in *y* (provided that no trailing blanks are entered):

```
>> SELECT NAME FROM EMPLOYEE
+>   WHERE NAME LIKE "%y@"
+>   TERMINATE "@" ;
NAME
----
Jay
Holly
```

- The character sets associated with the column, the LIKE pattern, the ESCAPE character, and the TERMINATE character must be the same.
- When two-character strings are considered equivalent to one-character strings, LIKE might not return the expected result. For example, if a-umlaut is considered equal to ae, LIKE a% does not return a match on string a-umlaut.

Examples—LIKE

- This LIKE predicate finds all employee names beginning with ZB:

```
EMPNAME LIKE "ZB%"
```

- This LIKE predicate finds all job titles that match a specific string provided at execution time:

```
JOB LIKE ?SOMEJOB
```

- This LIKE predicate finds all part descriptions that are not FLOPPY_DISK. The escape character indicates that the backslash in “FLOPPY_DISK” is part of the string to search for, not a wild-card character.

```
PARTDESC NOT LIKE "FLOPPY\_DISK" ESCAPE "\"
```

- This LIKE predicate specifies that the column NAME is associated with the ISO88591 character set. The predicate finds the value A_B:

```
NAME LIKE _ISO88591"A\_B" ESCAPE _ISO88591"\
```

Limits

NonStop SQL/MP has limits on the size and number of objects, on the size of columns, on file attributes, and on other items you use with NonStop SQL/MP.

SQL uses information stored in file labels and special records in the disk volume directory that contain information about a table, view, partition, or catalog. As a result, some limits depend on the size of file labels, which have a block size of 4096.

NonStop SQL/MP limits are summarized in this alphabetic list of items:

- Base tables per view

A view definition can include a maximum of 16 base tables in a FROM clause.

- Block size

The largest allowed block size is 4096 bytes.

- Clustering key length

The sum of the column lengths for the columns of the key cannot exceed 247 bytes. SQL appends an 8-byte SYSKEY column to the columns specified for the clustering key, making the maximum actual physical key length 255.

- Column heading

The SQLCI report writer imposes a maximum of 50 lines on a column heading. Any new-line character following the 49th one is text and does not create a new line. Column headings are defined with the HEADING clause of the CREATE or ALTER statements for a table or view.

- Columns per index

The maximum number of columns allowed for an index is 254 minus the number of columns in the primary key of the underlying table.

- Columns per table

The maximum number of columns allowed per table depends on the column definitions with these restrictions:

- The sum of the lengths of the columns in a row must not exceed the maximum row length (see the Row length item later in this subsection). In determining the column length for a varying-length character column, use the maximum declared length plus two bytes. For example, suppose that you declare a VARCHAR column of the maximum row length. The limit for the number of columns in this situation is one.

For each column that can contain null values, add two bytes to the column length.

- The description of the table must fit in the file label. The number of columns that fits depends on the data types of the columns and on the presence of

default values. File label restrictions can reduce the maximum number of columns allowed to as few as 128.

- Columns per view

The maximum number of columns allowed for a view is between 200 and 400, depending on the size of the column definitions. The size of the column definitions depends on the data types of the columns and the complexity and number of operators that define the selection of rows. The descriptions of the columns of the view must fit in a file label.

- Comments

A comment line in a catalog table cannot exceed 132 characters. The maximum number of lines allowed is 10,000.

- Constraint definition

The search condition that defines a constraint must be less than 3,000 bytes.

The code generated for the combined search conditions of all constraints associated with a table must be less than 31,000 bytes.

- Cursors

One object file can have up to a billion cursors. Assuming 300 bytes for each SQL statement object, 4 megabytes would hold 12,000 cursors. If the cursor definitions are complex or have large names, or there are many input or output variables, the limit becomes smaller.

- Data length

Fixed-length character data has a minimum length of one character. Varying-length character data has a minimum length of 0 characters, but the length field is two bytes. The maximum length of a character column depends on the file organization of the table that contains the column, on whether the character set associated with the column is a single or double-byte character set, and on whether the data type specifies a fixed or varying length.

Data Type	Key-Sequenced	Relative or Entry-Sequenced
Single-byte unvarying	4061	4072
Single-byte VARYING	4059	4070
Double-byte unvarying	2030	2036
Double-byte VARYING	2029	2035

A column that allows null values requires two extra bytes.

For clustering keys and system-defined primary keys, SQL requires 8 bytes for a system-defined key called SYSKEY.

A string literal can be as long as a character column. For more information, see [Data Types](#) on page D-1.

A numeric literal cannot exceed 18 digits.

- Default value for a character column

The maximum length for the default value is 8 characters.

- First key length

A specification for the first key for partitions cannot exceed the length of the user-defined primary key or clustering key for tables or the length of the index key for indexes. See the Index length item later in this subsection.

- FROM clause tables

The maximum number of tables that can be specified in a FROM clause is 16. This maximum includes the underlying base tables of views.

- IN predicate expressions

The maximum number of expressions in the *expression-list* of an IN predicate is 500.

- Index length

The maximum length depends on the index type:

Unique	The sum of the length of the columns in the index cannot exceed 253 bytes.
--------	--

Nonunique	The sum of the length of the columns in the index plus the sum of the key of the underlying table cannot exceed 253 bytes.
-----------	--

The length of any column that allows null values is 2 bytes longer than the declared length of the column.

- Indexes per table

The maximum number of indexes depends on the number of columns in the key, the number of partitions in each index, and additional factors. If there are no partitions in the indexes, a table can have a maximum of from 60 to 120 indexes; in some situations, however, as few as 32 indexes are allowed. If there are 25 partitions per index and there is a total of 10 columns in the primary key and index, the maximum number of indexes in a table could be reduced to as few as 34 (for a table with standard partition arrays) or 66 (for a key-sequenced table with extended partition arrays).

Additional factors that affect the index limit are the number of columns in the index, the number of indexes for the underlying table, the number of protection views defined for the underlying table, and the number of catalogs used to describe the indexes. The description of all the indexes for a table must fit in a file label.

- Lock limit

SQL defines a lock limit for the number of row locks that a program can own. The limit is internal to SQL/MP software. If a program exceeds the limit and the locks cannot be upgraded to table locks, the program receives an error message.

- **MAXEXTENTS**

The maximum number of extents allowed is 944 for a Format 1 partition. This number is decreased if the extent size (primary or secondary) is very large, such as a secondary extent size of 2,000 pages.

For Format 2 partitions, the maximum number of extents allowed is 919. This number can change per RVU and without notice.

- **Partition size**

For Format 1 partitions, the maximum partition size that SQL/MP software can support is approximately 2.1465 gigabytes. This number is determined by using this formula: $2^{31} - 1$ - 1 megabyte. The 1 megabyte of space is for internal use.

The formula for applying the limit to a partition is:

$$\text{PRIM} + (\text{SEC} * (\text{MAXEXT} - 1)) \leq 2.1465 \text{ gigabytes}$$

where

PRIM

is the primary extent size in bytes

SEC

is the secondary extent size in bytes

MAXEXT

is the maximum number of extents

For Format 2 partitions, the maximum partition size is limited to the size of the disk, up to one terabyte.

- **Partitions per index or table, extended partition array**

Typically, if the primary key of the index or table is between 10 and 50 bytes, the maximum number of partitions allowed ranges from approximately 900 (for the smaller key size) down to 400 (for the larger key size).

The maximum number of index partitions depends on the key size; if the primary key of a key-sequenced table is between approximately 10 and 50 bytes and the index key size is also between 10 and 50 bytes, the maximum number of partitions typically ranges from 674 (for keys of 10 bytes) to 246 (for keys of 50 bytes).

The actual calculation of the limit is complex and depends on such factors as disk label space and the message size of the NonStop operating system.

When you create or alter a table or index with a large number of partitions, the PARTNS catalog table and associated IXPART01 index might become full. To correct the situation, distribute object and partition definitions across multiple catalogs.

Actual limits depend upon the definition of the SQL tables and indexes, but the PARTNS and IXPART01 catalog tables can contain approximately 500,000 rows. Each table or index with N partitions stores N^2 rows of information in the PARTNS catalog table. Thus, three tables of 400 partitions each can be defined in a single catalog.

DDL and DML operations on tables or indexes that have large numbers of partitions might return file-system error 31 or 34 because of insufficient memory in the process file segment (PFS) used by the SQL file system. Actual limits depend upon the definition of the SQL tables and indexes and the SQL statement being executed, but memory limitations typically appear when a table or index has 400 or more partitions.

If you see one of these errors, you can increase PFS size using the `BINDER SET PFS integer` command:

```
1>RENAME SQLCAT,ZZSQLCAT
2>BIND
@ADD * FROM ZZSQLCAT
@SET PFS 256
@SELECT LIST (* OFF)
@BUILD SQLCAT !
@EXIT
3>FUP LICENSE SQLCAT
```

Alternately, for programs executed using TACL, you can specify PFS size in the TACL RUN command.

Increase PFS size selectively, only as the need arises. Keep the original copy of any program that requires a larger PFS setting. If you increase the PFS setting for SQLCAT or SQLUTIL, you must license the new copy. If you increase the PFS setting for SQLCI2, you must SQL compile the new copy.

- Partitions per index or key-sequenced table, standard partition array

Typically, if the primary key of the index or key-sequenced table is between 10 and 50 bytes, the maximum number of partitions allowed can be from approximately 230 (for the smaller key size) down to 110 (for the larger key size).

If the key of an index is between 10 and 50 bytes, the maximum number of index partitions is typically between approximately 180 (for keys of 10 bytes) and 60 (for keys of 50 bytes). In some situations, the maximum might be as few as 38.

Factors that affect the number of allowed partitions are the data types of the columns of the primary key and the number of catalogs used to describe the partitions. Also, the description of the partitioned table or index must fit in a file label.

- Partitions per index or key-sequenced table, format 2 enabled partition array

Typically, if the primary key of the index or key-sequenced table is between 10 and 50 bytes, the maximum number of partitions allowed can be from approximately 800 (for the smaller key size) down to 390 (for the larger key size).

If the key of an index is between 10 and 50 bytes, the maximum number of index partitions is typically between approximately 620 (for keys of 10 bytes) and 238 (for keys of 50 bytes).

The actual calculation of the limit is complex and depends on such factors as disk label space and the message size of the NonStop operating system.

When you create or alter a table or index with a large number of partitions, the PARTNS catalog table and associated IXPART01 index might become full. To correct the situation, distribute object and partition definitions across multiple catalogs.

Actual limits depend upon the definition of the SQL tables and indexes, but the PARTNS and IXPART01 catalog tables can contain approximately 500,000 rows. Each table or index with N partitions stores $N \times 2$ rows of information in the PARTNS catalog table. Thus, three tables of 400 partitions each can be defined in a single catalog.

DDL and DML operations on tables or indexes that have large numbers of partitions might return file-system error 31 or 34 because of insufficient memory in the process file segment (PFS) used by the SQL file system. Actual limits depend upon the definition of the SQL tables and indexes and the SQL statement being executed, but memory limitations typically appear when a table or index has 400 or more partitions.

- Partitions per relative or entry-sequenced table

This type of limit on partitions for a table is affected by the same restrictions that apply to key-sequenced tables with standard partition arrays. Additional restrictions also apply because of dependencies on the number and size of extents and on primary key values.

These details can help you estimate the maximum number of partitions allowed for relative or entry-sequenced tables:

- When you partition a table, NonStop SQL/MP evenly distributes all possible rows (identified by primary key value) into the partitions in both primary and secondary extents using the MAXEXTENTS attribute value to determine the number of extents and the EXTENTS attribute to determine extent sizes. (Relative files have SYSKEY values that begin at 0 and increment by 1. Entry-sequenced files have SYSKEY values based on the block number.)
- SQL assigns rows to partitions until it either runs out of primary key values or runs out of space declared for the table. You cannot specify partitions that would require primary key values greater than 4,294,963,199.
- The bigger your partitions are, the fewer you can specify.
- When you define a partition for a relative or entry-sequenced table, you do not specify the range of rows to be stored in the partition. SQL determines where rows are stored.

- Predicates per query

The maximum number of predicates allowed in an SQL/MP query is approximately 290, a Guardian operating system limitation. The exact limit depends on the combination of predicates and column data types.

- Prepared statements

You can have up to 20 prepared statements in an SQLCI session. (Programs can have more prepared statements.)

- Primary key

See [Syskeys](#) on page S-90, [Clustering Keys](#) on page C-28, or [User-Defined Keys](#) on page U-16.

- Row length

Row length is the sum of the lengths of the columns of a table. For each column that can contain null values, add 2 bytes to the column length when computing this sum. The length of a varying-length character column is its maximum declared length plus 2 bytes.

The maximum row length is the block size minus space for a file header. (The BLOCKSIZE attribute controls block size.)

A header is 32 bytes for key-sequenced tables and 22 bytes for relative and entry-sequenced tables. In addition, there are two bytes overhead for each row in a block.

For Format 2 key-sequenced tables, a file header is of 56 bytes.

- Statement length

The maximum length of an SQL/MP statement is 32,767 characters, including blanks.

- Subquery nesting

Queries can be nested a maximum of 16 levels, including the outermost query.

- SYSKEY value

The value range allowed for a SYSKEY (system-defined primary key) is 0 through 4,294,963,199 for the 4-byte primary key used in a table with relative or entry-sequenced file organization; the range is 0 through $2^{63}-1$ for the 8-byte primary key (actually, a timestamp) used in a table with key-sequenced file organization.

- Tables per query

The maximum possible number of tables in a database, if enough memory is available, is 32,767. Only 16 of these tables can be referred to in any given query.

- Temporary file size

The size of a temporary file is limited to the space available on the disk on which the file resides. Temporary files provide space for sort operations required for some queries, for creating indexes, for splitting partitions, and for loading tables.

- Transactions per table

The limit on the number of TMF transactions or update transactions that can be active on a given SQL table is the same as the limit on the number of transactions that can be active in your NonStop system. That number is configurable and depends on the TRANSPERCPU attribute of the BEGINTRANS object of TMF. For more information, see the description of the ALTER BEGINTRANS command in the *TMF Reference Manual*.

- User-defined primary key length

The sum of the column lengths for the columns of the key cannot exceed 255 bytes.

- User process sort

No more than 32,767 rows can be sorted.

- View definition

The CREATE VIEW statement, including any name expansion from the use of asterisks in column, view, and table specifications, can have a maximum of 3,000 bytes.

- Views per table

Approximately 180 protection views can be defined for a table. The limit is determined by the requirement that a small amount of information about each protection view defined on a table must fit in a file label.

There is no maximum number of shorthand views for a table.

LINE_NUMBER Function

LINE_NUMBER is an SQLCI report writer function that returns the line number of the current detail line. LINE_NUMBER is useful for numbering detail lines in a report.

You can use LINE_NUMBER in any report writer command with a print list, but SQL calculates the function value for detail lines only (not title or footing lines, for example), so it is generally useful only in DETAIL commands.

```
LINE_NUMBER [ OVER REPORT      ]
            [ OVER PAGE        ]
            [ OVER break-column ]
```

The default is OVER REPORT.

OVER REPORT

determines the line number by setting a count of 0 at the beginning of the report and incrementing the number by 1 at the start of each detail line.

OVER PAGE

determines the line number by setting a count of 0 at the beginning of each page and incrementing the number by 1 at the start of each detail line. (A detail line is a logical output line specified by the print list on a DETAIL command; it might consist of more than one physical line.)

OVER *break-column*

determines the line number by setting a count of 0 when the value of the specified break column changes and incrementing the number by 1 at the start of each detail line in the group. *break column* is a column name, alias, detail alias, or COL *number* that identifies a column named in a BREAK ON command.

Considerations—LINE_NUMBER

- The default format for line numbers is I11.
- If you specify LINE_NUMBER OVER *break-column*, you must enter a BREAK ON command that defines the referenced break column (and that also defines break columns referenced in other LINE_NUMBER function calls in the current set).

Examples—LINE_NUMBER

- This example produces a line that contains a line number, part number, and part name. The line numbers start over at 1 on the first line of each page.

```
S> DETAIL LINE_NUMBER OVER PAGE, PARTNUM, PARTDESC;
(EXPR)      PARTNUM  PARTDESC
-----
          1         212  SYSTEM 192KB CORE
          2         244  SYSTEM 192KB SEMI
          3        1403  PROC    96KB SEMI
          ...

(EXPR)      PARTNUM  PARTDESC
-----
          1         2053 EDITOR
          2         2267 TEXT FORMATTER
          3         2598 C COMPILER
          ...
```

- This example prints the same information as the preceding command but in a different format.

```
S> DETAIL LINE_NUMBER AS I4 NOHEAD, PARTNUM, PARTDESC;

PARTNUM  PARTDESC
-----
1         212  SYSTEM 192KB CORE
2         244  SYSTEM 192KB SEMI
3        1403  PROC    96KB SEMI
          ...
```

LINE_SPACING Option

LINE_SPACING is an option of the SQLCI report writer SET LAYOUT command that specifies how many lines to advance between report lines.

LINE_SPACING also defines the increment of the SKIP clause. For example, if you set the LINE_SPACING option to 2 and specify SKIP 3 as a print item in a print list, the report writer skips six (2*3) lines before printing the next report line.

```
LINE_SPACING number
```

The default is 1. (Single spacing)

number

is an integer in the range 1 through 32,767 that specifies how many lines to advance before printing the next report line.

Examples—LINE_SPACING

- This example sets double spacing:

```
>> SET LAYOUT LINE_SPACING 2;
```

- This example sets triple spacing and page length 62:

```
>> SET LAYOUT PAGE_LENGTH 62, LINE_SPACING 3;
```

LIST Command

LIST is an SQLCI report writer command that displays rows returned by the SELECT command. You can use LIST only from the select-in-progress prompt (S>).

```
L[IST] { { F[IRST] } [ number ] }
      { { N[EXT] } }
      { A[LL] } ;
```

F[IRST] [*number*]

displays the first *number* rows of the result table and then returns to the select-in-progress prompt. If you omit *number*, SQL uses the current value of the LIST_COUNT session option. (The default for the LIST_COUNT session option is ALL; you can change it with the SET SESSION command.)

N[EXT] [*number*]

displays the next *number* rows of the result table and then returns to the select-in-progress prompt (S>).

If you omit *number*, SQL uses the current value of the LIST_COUNT session option.

Using LIST NEXT without *number* is equivalent to pressing the return key at the S> prompt.

A[LL]

displays all rows of the result table and returns you to the standard SQLCI prompt (>>).

Considerations—LIST

- Rows are listed on the OUT_REPORT file, or (if you did not specify one) on the OUT file. The default for OUT and OUT_REPORT file is the home terminal of your SQLCI process, which is typically your terminal.
- If you are experimenting with a report format, set the LIST_COUNT option to a small number. Use LIST ALL only when you are ready to print the final report.

Example—LIST

This example sets the LIST_COUNT option to 2, issues a SELECT statement that displays the first rows returned (because LIST_COUNT is two), and then uses the LIST command to display the next five rows:

```
>> SET LIST_COUNT 2;
>> SELECT DISTINCT PARTS.PARTNUM, PARTDESC, QTY_ON_HAND,
+> LOC_CODE, PRICE
+> FROM SALES.PARTS, INVENT.PARTLOC
+> WHERE PARTS.PARTNUM = PARTLOC.PARTNUM
+> ORDER BY PARTS.PARTNUM;
PARTNUM  PARTDESC                QTY_ON_HAND  LOC_CODE  PRICE
-----  -
      212  PC SILVER, 20 MB          18  A87          2500.00
      212  PC SILVER, 20 MB          20  G87          2500.00
S> L N 5;
      244  PC GOLD, 30 MB          23  P78          3000.00
      244  PC GOLD, 30 MB          43  A78          3000.00
      255  PC DIAMOND, 60 MB        21  A21          4000.00
      2001  GRAPHIC PRINTER,M1         0  P10          1100.00
      2001  GRAPHIC PRINTER,M1        800  A10          1100.00
```

Literals

Literals are numeric, string, date-time, or INTERVAL constants that can be used in expressions, in statements, or as parameter values.

For more information, see the entries for specific types of literals:

[Date-Time Literals](#)

[INTERVAL Literals](#)

[Numeric Literals](#)

[String Literals](#)

LOAD Command

LOAD is an SQLCI utility command that loads data from an SQL table or a Guardian file (such as a Guardian process, device, unstructured disk file, or non-SQL object) into either an SQL table and its indexes or an Enscribe structured disk file. LOAD overwrites existing data in the target table or file.

LOAD resembles the FUP LOAD command but, unlike FUP LOAD, it works with SQL objects.

If SMF is installed on your node, LOAD syntax cannot specify any file located on a `$.ZYS*` subvolume.

-
- △ **Caution.** When loading an entire table, LOAD requires that you turn off auditing for the table. This requirement invalidates TMF online dumps of the table and its indexes. To ensure TMF volume recovery protection for the table and its indexes, turn AUDIT back on when the LOAD is complete, and make new TMF online dumps of all partitions of the table and its indexes.
-

When using the PARTONLY option to load a single partition, you do not need to turn off auditing, as the command does it for you. You still need to make an online dump of the single partition.

```
LOAD in-file, out-file [ [, load-option ] ... ;
```

load-option is:

```
{ control-option                                }
{ in-option                                     }
{ key-sequence-option                          }
{ move-option                                  }
{ PARALLEL EXECUTION { ON [config-op] | OFF } }
```


control-option is:

```
{ ALLOWERRORS [ ON | OFF | num ]
  COUNT num-records
  EMPTYOK
  FIRST { ordinal-record-num
          KEY record-spec
          KEY (key-value[, key-value] ... )
          key-specifier ALTKEY (key-value
                                [, key-value] ... )
        }
  PAD pad-character
  REPLACE SPACES WITH { ZERO[ES] | DEFAULT[S]}
  UNSTRUCTURED
  UPSHIFT
  USESQLNULLS
  SQLNULLABLE
}
```

in-option is:

```
{ BLOCKIN in-block-length
  { COMPACT | NO COMPACT }
  EBCDICIN
  RECIN in-record-length
  REELS num-reels
  { REWINDIN | NO REWINDIN }
  SHARE
  SKIPIN num-eofs
  TRIM trim-character
  { UNLOADIN | NO UNLOADIN }
  VARIN
}
```

key-sequence-option is:

```
{ { PARTONLY
  { PARTOF volume-name }
  PARTONLYIN
  SORTED
  MAX num-records
  SCRATCH scratch-file
  DSLACK percent
  ISLACK percent
  SLACK percent
}
```

move-option is:

```
{ SOURCEDICT dictionary-name }
{ SOURCEREC  record-name      }
{ TARGETDICT dictionary-name }
{ TARGETREC  record-name      }

MOVE { source-name TO target-name }
      { ( source-name TO target-name }
      {   [, source-name TO target-name]... ) }

MOVEBYNAME [ ON | OFF ]
MOVEBYORDER [ ON | OFF ]
TRUNC[ATION] [ ON | OFF ]
REDEFINE ( redefine-spec [, redefine-spec]... ) }
```

redefine-spec is:

original-qualified-name AS *redefined-qualified-name*

config-op is:

```
CONFIG { config-file [ FOR index-name ] }
        { ( config-file FOR index-name }
        {   [, config-file FOR index-name]... ) }
```

in-file

is the name (or an equivalent DEFINE) of the table or file from which to load data. *in-file* can be a Guardian process, device (such as a terminal or tape), unstructured disk file, or non-SQL object.

out-file

is the name (or an equivalent DEFINE) of an existing SQL table or non-SQL object to load.

ALLOWERRORS [ON | OFF | *num*]

specifies what happens when errors occur.

ON skips nonconvertible records but process subsequent records.

OFF stops the load operation after the first conversion error.

num skips nonconvertible records until the number of such records exceeds the value of *num*. The maximum value for *num* is 32,767.

If you omit the ALLOWERRORS clause completely, the default is ALLOWERRORS OFF. If you specify ALLOWERRORS but do not specify an option, the default is ALLOWERRORS ON.

Nonconvertible records include records that contain one or more of:

- A nonnumeric value in a numeric field
- A duplicate key value in the primary key field of the output file
- A null value when the target field cannot represent a null value
- Parity errors
- An ordering that does not match the sort criteria for the output file (only if you specified SORTED)
- Inconsistencies with constraints defined for the output table

For more information about the rules for data conversion, see [CONVERT Command](#) on page C-94.

COUNT *num-records*

specifies the number of records to load. The default is all records.

EMPTYOK

directs LOAD to accept an empty input file that results in an empty output file. If the input file is empty and you do not specify EMPTYOK, LOAD terminates and reports an error without overwriting an existing output file.

```
FIRST { ordinal-record-num
      { KEY record-spec
      { KEY (key-value[, key-value] ... )
      { key-specifier ALTKEY ( key-value
      { [ , key-value ] ... ) }
```

specifies the starting record of the input file from which to begin loading. If you omit FIRST, the load operation starts with the first record.

ordinal-record-num

is the number of records (from the beginning of the file) to skip. If you specify this option for an unstructured disk file, the loading begins at this offset:

ordinal-record-num * *in-record-length*

KEY *record-spec*

specifies the primary-key value for the starting record of an unstructured file, or of a relative or entry-sequenced file or table. The record you name with KEY is the first record to load. Specify *record-spec* as an integer from 0 through 4,294,967,295.

- For unstructured files, *record-spec* is the starting relative byte address.
- For relative files or tables, *record-spec* is the starting record number.

- For entry-sequenced files or tables, *record-spec* is the Enscribe record address. For descriptions about the record addresses in entry-sequenced files and tables, see the *Enscribe Programmer's Guide*.

KEY (*key-value*> [, *key-value*] ...)

indicates the approximate position of the starting record for key-sequenced files. Subsequent rows are obtained in primary key order. *key-value* is either a string in quotation marks or an integer in the range 0 to 255. (Each integer represents the value of 1 byte.)

For more information about specifying a FIRST KEY value, see [Considerations—LOAD](#) on page L-33.

key-specifier ALTKEY (*key-value* [, *key-value*] ...)

indicates the approximate position within the specified alternate key file of the starting record for key-sequenced files. Subsequent rows are obtained in alternate key order.

key-specifier is a one-character or two-character string in quotation marks, or a numeric literal in the range 0 through 32,767 that designates the alternate key to use for the positioning.

Specify *key-value* for key-sequenced files according to the description of *key-value* in the preceding KEY (*key-value*) option and under [Considerations—LOAD](#) on page L-33.

PAD *pad-character*

(for loading non-SQL objects only) pads output records containing less than *in-record-length* bytes with the *pad-character* up to the record length specified in the file label of the output file. Specify *pad-character* as a single ASCII character inside quotation marks ("c") or as a numeric literal in the range of 0 through 255 representing the byte value of the character.

REPLACE SPACES WITH { ZERO[ES] | DEFAULT[S] }

specifies how to load an Enscribe ASCII numeric decimal field that contains all blanks into an SQL numeric column. ZEROES specifies that the target column should be set to zero; DEFAULTS specifies that the target column should be set to its default value. This option does not apply to Enscribe numeric binary fields.

If you do not specify this option for an Enscribe ASCII numeric decimal field, a conversion error occurs for any record in which the field contains blanks.

UNSTRUCTURED

(for loading from a table or disk file only) directs LOAD to handle the data as a sequence of bytes, ignoring any record structures normally recognized for the table or file. This option is typically used with the COPY command to let you examine

raw data on disc. You do not need to request the UNSTRUCTURED option to examine an unstructured file.

UPSHIFT

(for loading a non-SQL object only) converts all bytes of the input that contain lowercase ASCII characters to uppercase ASCII characters before loading the data to the target record.

The UPSHIFT conversion is made without regard to the data types of fields or columns of the input, so undesired changes to the data can occur if you use UPSHIFT with input that is not composed of simple character data.

Although you cannot specify the UPSHIFT option if *out-file* is an SQL table, data moved to an SQL column that has the UPSHIFT attribute is automatically upshifted.

USESQLNULLS

(for loading from non-SQL objects to SQL tables only) specifies that a null value from a non-SQL object be loaded as an SQL null value.

USESQLNULLS applies only if the SQL column being loaded allows null values, if you also specify the SOURCEREC option, and if the Enscribe null value appears in every byte of the Enscribe field. (Enscribe allows you to specify a character to use as the null character for a field at file-creation time, and then uses that character to represent null values within the field. Any field that is filled entirely with the null character is handled as null.)

If you omit USESQLNULLS, LOAD provides no special treatment for Enscribe null characters.

SQLNULLABLE

specifies that the support for null indicator values for Enscribe fields is enabled when loading data from Enscribe to SQL or from SQL to Enscribe.

The SQLNULLABLE option causes LOAD to look for the SQLNULLABLE DDL clause in the TARGETREC or SOURCEREC option, and depending on this clause, add or read a two-byte null indicator for the Enscribe field value. Without TARGETREC and SOURCEREC options, LOAD adds or reads the null indicator for all Enscribe fields that correspond to a nullable column in the SQL table.

If you omit the SQLNULLABLE clause, LOAD does not support null indicators for Enscribe fields. SQLNULLABLE does not affect loading from SQL tables to SQL tables and Enscribe files to Enscribe files. The SQLNULLABLE option for Enscribe to SQL loading automatically enables the USESQLNULLS option.

in-option

specifies characteristics of the input file.

`BLOCKIN in-block-length`

specifies the length of an input block in bytes. *in-block-length* is a value from 1 through 32,767 that indicates the actual number of bytes requested in a single physical read operation.

BLOCKIN does not apply to a table unless you specify the UNSTRUCTURED option. If the input block length exceeds the input record length specified in the `RECIN in-record-length` option, input records are deblocked. Records of the specified length are extracted from the input block until the number of bytes extracted equals the block length or until the last input record is read.

The read record count for all but possibly the last record in a block is equal to *in-record-length*. If the input block length is not an even multiple of *in-record-length*, the last record extracted from a full block is a short record with a read count equal to the number of bytes extracted.

If you omit the BLOCKIN option and *in-file* is not a labeled tape, SQL uses the RECIN record length for the block length and reads each input record in a separate physical operation.

If *in-file* is a labeled tape, you can specify the input block length with either the BLOCKIN clause of the LOAD command (as described here) or with the BLOCKLEN attribute of the CLASS TAPE DEFINE for the tape. If you specify values for both the BLOCKIN clause and the BLOCKLEN attribute, the values must match.

`{ COMPACT | NO COMPACT }`

(for loading from files or tables with relative file organization only) controls whether zero-length (empty) records on files or tables with relative file organization are ignored when a file is read. COMPACT ignores empty records and renumbers records that follow an empty record; NO COMPACT copies empty records. The default is COMPACT.

For information about the impact of changes in SYSKEY, see [Syskeys](#) on page S-90.

`EBCDICIN`

(for loading from non-SQL objects only) translates EBCDIC characters to their ASCII equivalents in the input file. If you omit EBCDICIN, LOAD does not translate input.

In a conversion between ASCII and EBCDIC, the symbols representing each character are the same in ASCII and EBCDIC except for:

ASCII	EBCDIC
Exclamation point	Logical OR

ASCII

Left square bracket
 Right square bracket
 Circumflex

EBCDIC

Cent sign
 Exclamation point
 Logical NOT sign

The conversion is done without regard to the data types of fields or columns of the input, so undesired changes to the data can occur if you use EBCDICIN with input that is not composed of simple character data.

`RECIN in-record-length`

specifies the maximum length of an input record in bytes. The actual number of bytes in each input record (the read count) depends on whether you also specify TRIM:

- If you do not specify TRIM, the read count is the actual number of bytes in the input record. (For unstructured files that are not EDIT files, the read count is exactly *in-record-length* bytes, although fewer bytes might be read from the last record; for all other files, the read count is the number of bytes actually read.)
- If you specify TRIM, every trailing *trim-character* is deleted from the input record. The read count includes only the characters that have not been trimmed.

If you omit RECIN, SQL determines *in-record-length*:

- If you specify *in-block-length* as less than or equal to 4096, the value of *in-record-length* is used for *in-block-length*. If *in-block-length* is greater than 4096, *in-record-length* is 4096.
- If you do not specify *in-block-length*,
 - If *in-file* is a structured disk file or a nondisk device, *in-record-length* is the record length specified or calculated when the file was created (or when the system was generated).
 - If *in-file* is an unstructured disk file, and if *out-file* is a table and you do not specify VARIN, *in-record-length* is the length of the logical record specified by SOURCEREC or—if you do not specify SOURCEREC—the length of the logical record implied by the description of the output table; otherwise, *in-record-length* is 132.
 - If *in-file* is a process, *in-record-length* is 132.

RECIN does not apply to a table unless you specify the UNSTRUCTURED option.

If *in-file* is a labeled tape, you can specify the input record length with either the RECIN clause of the LOAD command (as described here) or with the RECLen attribute of the CLASS TAPE DEFINE for the tape. If you specify values for both the RECIN clause and the RECLen attribute, the values must match.

REELS *num-reels*

(for loading from an unlabeled magnetic tape only) sets the number of reels that make up the input file. Specify *num-reels* as an integer from 1 through 255. The tape is read until two consecutive EOF marks are reached for *num-reels*. At each end of reel before the last reel, the tape is rewound and unloaded, and you are prompted for the next reel. If you omit REELS, *in-file* data transfer terminates when a single EOF mark is encountered.

{ REWINDIN | NO REWINDIN }

(for loading from a magnetic tape only) specifies whether the tape is rewound when the end of file is read from tape. If you specify NO REWINDIN, the tape remains positioned. The default is REWINDIN.

SHARE

(for loading from disk only) opens *in-file* in shared exclusion mode. Using SHARE, you can access an *in-file* even if it is currently open by another process, unless that process is open with exclusive exclusion mode. If you omit SHARE and *in-file* is a table or disk file, LOAD opens *in-file* with protected exclusion mode. The SHARE option cannot use certain internal performance features and, therefore, might perform more slowly than a LOAD request without the SHARE option.

SKIPIN *num-eofs*

(for loading from an unlabeled magnetic tape only) moves the tape specified as *in-file* past *num-eofs* end-of-file marks before starting to transfer data. Specify *num-eofs* as an integer from -255 through 255.

- If you specify a positive value for *num-eofs*, the tape is wound forward past *num-eofs* EOF marks and is positioned immediately after the last EOF mark passed.
- If you specify a negative value for *num-eofs*, the tape is wound backward over (-1 times *num-eofs*) EOF marks, then moved forward and positioned immediately ahead of the last EOF mark passed.
- If you specify a value of zero for *num-eofs*, the SKIPIN option is ignored.

If you omit the SKIPIN option, the tape remains at its current position, and the data transfer begins with the next physical record on tape.

TRIM *trim-character*

(for loading from a non-SQL object only) deletes any trailing characters in each record matching *trim-character*. Specify *trim-character* as a single ASCII character inside quotation marks ("c") or as a numeric literal in the range of 0 through 255 representing the byte value of the character.


```
{ UNLOADIN | NO UNLOADIN }
```

(for loading from a magnetic tape only) specifies whether the tape is unloaded when rewinding occurs. The default is UNLOADIN (the tape is unloaded when rewound).

VARIN

(for loading only from files—not tables—with variable-length, blocked records, such as those produced using the VAROUT option of COPY) tells LOAD that *in-file* contains variable-length, blocked records that begin with a word that contains the length of the record; the read count equals the value of that length indicator.

key-sequence-option

specifies how to load tables or files with key-sequenced file organization.

```
{ PARTONLY  
{ PARTOF volume-name }
```

(for loading files or tables with key-sequenced file organization only) directs SQL to load only a single partition. PARTONLY affects only the output and has no effect on how SQL reads the input file.

For an SQL table, PARTONLY directs SQL to load the partition specified as *out-file* while PARTOF directs SQL to load the partition on volume *volume-name*. If you specify PARTOF, you can specify any partition of the table as *out-file*.

For a non-SQL object, PARTOF directs SQL to load the partition specified as *out-file*. *volume-name* must specify the volume that contains the primary partition. You cannot use PARTONLY with a non-SQL object.

You cannot use PARTONLY or PARTOF if the table has indexes defined for it. You cannot use PARTONLY for an audited table if BLOCKSIZE is less than 2KB.

PARTONLYIN

(for loading files or tables with key-sequenced file organization only) directs SQL to load only from a single partition. PARTONLYIN affects only the input and has no effect on how SQL writes the output file.

For an SQL table and non-SQL object, setting the PARTONLYIN option directs SQL to load from the partition specified as *in-file*.

SORTED

(for loading files or tables with key-sequenced file organization only) specifies that input file records are already sorted in the key-field order of the output file and are not to be resorted. If you omit this option and the target file is key-sequenced, LOAD sorts the records before loading the output file.

MAX *num-records*

(for loading files or tables—but not indexes—with key-sequenced file organization only) specifies the number of input records as an integer from 0 through 2,147,483,647.

LOAD uses *num-records* to determine the size of the scratch file to be used by the SORT process. If you specify the SORTED option, you do not have to specify the MAX option.

If you underestimate the number of records, the sort can be significantly slower. If you overestimate the number of records, the cost is small.

The default is MAX 50000 unless an `=_SORT_DEFAULTS DEFINE` with VLM ON is in effect. With VLM ON, the default is MAX 1000000. For more information about VLM, see [=_SORT_DEFAULTS DEFINE](#) on page Z-4.

For loading indexes, LOAD estimates the maximum number of input records based on the size of the base table, ignoring any value you specify for MAX.

SCRATCH *scratch-file*

(for loading only files or tables with key-sequenced file organization) identifies the file to be used for temporary storage by the SORT process. *scratch-file* is a Guardian name.

If you omit this option, LOAD creates and uses a scratch file on a volume FastSort chooses by its characteristics. The default initial scratch volume is usually the volume where the SORTPROG program file resides. To override the automatic selection algorithm, specify an initial scratch file or volume in the SCRATCH attribute of a `=_SORT_DEFAULTS DEFINE`.

When loading a very large table, you might want to use a partitioned scratch file. Use the FUP CREATE command to create the scratch file and identify the file to LOAD with the SCRATCH option or the `=_SORT_DEFAULTS DEFINE`.

If you specify the SORTED option, you do not have to specify the SCRATCH option.

For more information on how to specify and manage scratch files, see the *FastSort Manual*.

DSLACK *percent*

(for loading only files or tables with key-sequenced file organization) specifies the minimum percentage of space to be left in data blocks for future insertions. Specify *percent* as a numeric literal from 0 through 99. If space is not available when an insertion is made, a block split occurs.

If you omit this option, LOAD uses the SLACK percent value.

ISLACK percent

(for loading only files or tables with key-sequenced file organization) specifies the minimum percentage of space to be left in index blocks for future insertions. Specify *percent* as a numeric literal from 0 through 99. If space is not available when an insertion is made, a block split occurs.

If you omit this option, LOAD uses the SLACK percent value.

SLACK percent

for key-sequenced targets only, specifies the minimum percentage of space to be left in both index blocks and data blocks for future insertions. Specify *percent* as a numeric literal from 0 through 99. If space is not available when an insertion is made, a block split occurs.

If you omit this option, LOAD leaves 15 percent slack space in both the data and index blocks.

If you specify DSLACK, ISLACK, and SLACK, the LOAD utility uses the DSLACK value for data blocks and the ISLACK value for index blocks. The SLACK value is ignored.

move-option

specifies names of elements related to the table or file and how to map source names to different target names.

SOURCEDICT dictionary-name

identifies the subvolume containing the DDL dictionary that contains the record or DEF definition for an Enscribe input file. *dictionary-name* is a Guardian subvolume name.

If you omit the SOURCEDICT option, LOAD assumes that the DDL dictionary resides on the current default subvolume.

SOURCEREC ddl-record-name

identifies the name of the DDL record or DEF definition for an Enscribe input file. *ddl-record-name* must be a valid DDL data name.

If you omit the SOURCEREC option, LOAD assumes that fields in the file occur in the same order as the columns in the target table and that all variable-length character fields are expanded to the maximum length and padded with blanks. For information about how LOAD converts other data types if they are not explicitly defined, see the Data type compatibility and field conversions item under [Considerations—LOAD](#) on page L-33.

TARGETDICT *dictionary-name*

specifies the subvolume containing the DDL dictionary that contains the DDL record or DEF definition for an Enscribe output file. *dictionary-name* is a Guardian subvolume name.

If you omit the TARGETDICT option, LOAD assumes that the dictionary resides on the current default subvolume.

TARGETREC *record-name*

(for loading non-SQL objects only) specifies the DDL record name of the record or DEF definition for *outfile*.

If you omit the TARGETREC option, SQL assumes that fields in *out-file* occur in the same order as fields or columns in *in-file* and that all variable-length character fields are expanded to the maximum length and padded with blanks. For information about how LOAD converts other data types if they are not explicitly defined, see the Data type compatibility and field conversions item under [Considerations—LOAD](#) on page L-33.

```
MOVE { source-name TO target-name
      { (source-name TO target-name
        { [,source-name TO target-name ] ... ) }
```

associates a field of the source with a field in the target so that data from the source field is loaded into the specified target field. Any column except a system-defined primary key can be a source or target item.

source-name

is the name of a DDL elementary data field item if you are loading data from a non-SQL object, or a column name if you are loading data from a table.

target-name

is the name of a DDL elementary data item if you are loading data to a non-SQL object, or a column name if you are loading data to a table.

You cannot specify a DDL group name (except the VARCHAR group) or the name of an array of fields in the MOVE option. You can, however, subscript or qualify the name of a DDL elementary data item. You need to qualify the name if it is ambiguous.

You cannot specify a system-defined primary key in the MOVE option.

MOVEBYNAME [ON | OFF]

specifies whether or not each field in the source record that has the same name as a field in the target record is copied to the target field with the same name.

The DDL hyphen (-) and the SQL underscore (_) are equivalent.

You can specify both MOVE and MOVEBYNAME. For fields where the two options conflict, MOVE overrides MOVEBYNAME.

If MOVEBYNAME is ON, a DDL group representing a variable-length character string can be loaded. For more information, see [Conversion of DDL Elementary Items](#) on page C-103 under CONVERT.

MOVEBYNAME OFF is the default.

MOVEBYORDER [ON | OFF]

specifies whether fields in the source are loaded to fields in the target on the basis of position. Data is loaded from the first field of the source record to the first field of the target record (row), from the second source field to the second target field, and so forth.

If you specify MOVEBYORDER ON, you cannot specify either the MOVE option or MOVEBYNAME ON.

If MOVEBYORDER is ON, a DDL group representing a variable-length character string can be loaded. For more information, see [Conversion of DDL Elementary Items](#) on page C-103.

MOVEBYORDER ON is the default.

TRUNC[ATION] [ON | OFF]

specifies whether to truncate data from a source field that is longer than its target field.

If TRUNC OFF is in effect, LOAD reports an error if a source field is longer than its target field.

TRUNC OFF is the default.

REDEFINE (*redefine-spec* [, *redefine-spec*] ...)
redefine-spec is:

original-qualified-name AS *redefined-qualified-name*

specifies that original DDL items (groups or fields) are loaded to columns based on redefinitions of the items. If you load from a table to a file, columns are loaded to DDL items based on redefinitions.

Unless you include the REDEFINE option, all redefined items are loaded according to the original definition of the original items.

original-qualified-name

identifies an original field or group in a DDL record. The name must be qualified by the group names at all preceding levels. For example,

CUSTOMER.ADDRESS.STREET-ADDRESS

is the qualified name of the STREET-ADDRESS field in the ADDRESS group.
The ADDRESS group is in the CUSTOMER group.

redefined-qualified-name

identifies a redefined field or group that corresponds to the original field or group.
The name must be qualified by the group names at all preceding levels. For example,

CUSTOMER.ADDRESS.STREET-DETAIL

is the qualified name of the STREET-DETAIL field that redefines the STREET-ADDRESS field.

When the field or group is loaded, the load is based on the redefinition you specify.

PARALLEL EXECUTION { ON [*config-op*] | OFF }
config-op is:

```
CONFIG { config-file [ FOR index-name ]
        {
          ( config-file FOR index-name
            [, config-file FOR index-name]... )
        }
```

specifies whether to load partitions of a partitioned index in parallel. The default (PARALLEL EXECUTION OFF) is to load partitions sequentially.

PARALLEL EXECUTION ON applies only to one partitioned index at a time. If the loaded table has more than one partitioned index, the partitions of the first index are loaded in parallel first. After the first index has been loaded, the partitions of the second index are loaded in parallel, and so forth. The parallel load index operation, therefore, loads the partitions of a physical index simultaneously; it does not load two physical indexes simultaneously.

config-file is the name of an EDIT file that contains instructions for configuring the processes that load the index. For information about how to specify configuration instructions in *config-file*, see [Parallel Index Loading](#) on page P-5.

FOR INDEX *index-name* specifies which index to load in parallel using the configuration in the preceding *config-file*. *index-name* is the name of any partition of the index. Each specified index must be unique.

If you specify only one configuration file in the PARALLEL EXECUTION clause and omit the FOR *index-name* clause, the configuration file applies to all indexes for the loaded table.

If you specify more than one configuration file in the PARALLEL EXECUTION clause, you must specify the FOR *index-name* clause for each configuration file. If at least one index is specified in the CONFIG clause, the partitions of any indexes not specified are loaded in parallel using default configuration values.

Considerations—LOAD

- LOAD requires authority to read the source file and to write to the target file. If you load data to or from a table, you must have authority to read the catalog in which the table is described. If the target file is a table, LOAD requires the authority to write to the catalog in which the table is described.
- LOAD first purges all data from the target file or table, then begins writing source records or rows to the target file or table, converting data and reorganizing records or rows as appropriate.

For full-table loads, LOAD sets the corrupt flag on the base table and indexes before the load starts. If the operation finishes successfully, LOAD resets the flags. If the operation fails, the corrupt flag indicates that the file is unusable.

If the target is a table, LOAD automatically loads any indexes on the table after it loads the table.

When you load a very large key-sequenced file and the data must be sorted, you might want to use a partitioned scratch file. Use FUP CREATE to create the scratch file and the SCRATCH option of LOAD to identify it.

If the input records must be sorted, disk space for the sort scratch file and for the output file must exist concurrently during the sorting phase.

- LOAD resembles COPY in that both transfer data from an existing source to an existing target. The major differences between LOAD and COPY are:
 - LOAD is typically used to enter initial data into an empty file. COPY is typically used to add data to a file that already contains data.
 - LOAD erases or overwrites existing records. COPY does not erase or overwrite existing records.
 - LOAD does not write to unstructured files or non-disk files but COPY does.
 - COPY and LOAD provide the same *in-options* and *move-options*, but LOAD provides additional *key-sequence-options* for loading key-sequenced files. These options let you load single partitions, sort output, and specify the amount of slack to leave in index and data blocks.
 - COPY is slower than LOAD.
- You cannot execute LOAD within a user-defined TMF transaction.

You cannot load data into an audited table. If the table to be loaded is audited, you must turn auditing off for this table before executing the LOAD utility. If you load a partitioned table using the PARTONLY option, this restriction does not apply.

You should perform these operations when you need to load data into an audited table:

1. Use ALTER TABLE to set the AUDIT attribute of the table OFF.
2. Execute the LOAD utility to load data into the nonaudited table.
3. Use ALTER TABLE to set the AUDIT attribute of the table ON.
4. Perform online dumps of all table partitions and indexes. For more information about performing online dumps, see the *TMF Operations and Recovery Guide*.

If you press the Break key to interrupt a load operation, SQLCI stops the load operation and returns the SQLCI prompt immediately, but any sort processes started by the load operation continue to execute to completion unless you exit SQLCI and stop them separately. Any data loaded at the time you interrupt a load operation remains in the table or file.

If a load operation fails, the target table or file is not usable. (Depending upon the reason for the failure, it might or might not be empty and it might or might not be marked corrupt, but in no case is it usable.) Make any necessary corrections in the LOAD options you specified and rerun the load operation. Alternately, if you do not want to rerun the load operation, use the PURGEDATA command to clear the corrupt flag.

LOAD errors can occur at the time LOAD compiles the operation or at the time the actual data is loaded. Compile-time errors (such as references to nonexistent fields, mapping between incompatible source and target fields, or unmapped target fields that have no default value) always terminate the LOAD operation. Data-loading errors (such as violations of integrity constraints, source data that exceeds the size of the target field, or a null value in a field mapped to a target field that does not allow null values) can be permitted or restricted with the ALLOWERRORS option.

- Rules for using LOAD with CLASS TAPE DEFINES or labeled tapes are described in the discussion of the FUP COPY command in the *File Utility Program (FUP) Reference Manual*.
- Loading data into Enscribe files and other non-SQL objects

A non-SQL object into which you load data must be an existing file. LOAD does not load data into any alternate-key file associated with a non-SQL object.

The target file must have default values defined for fields that do not have source fields mapped to them. To define a default value for a data item, use the VALUE IS clause in the DDL description of the field.

When you load key-sequenced partitioned files, consider:

- In a partitioned file, the range of keys for the different partitions is stored in the primary partition.

- To load a single partition (primary or secondary), specify the name of the partition as *out-file*, and specify the name of the volume that contains the partition in the PARTOF option. If you attempt to load a secondary partition when you have not specified the PARTOF option, you receive an error message.
- To load all partitions, specify the name of the primary partition as *out-file* and omit the PARTOF option.
- Be careful when you use PAD and TRIM options. If your data contains the *trim-character* or *pad-character*, data might be added or lost. Use a *pad-character* or *trim-character* that does not appear in your data. For example, suppose that you pad each record in a data file with zeros to a standard size in bytes and then store the records in another file. If you later trim the trailing zeros when you load the stored records, zeros at the end of the original data are also trimmed.
- The target file must have default values defined for columns that do not have source fields mapped to them. To define default values, use the DEFAULT clause of the CREATE TABLE or ALTER TABLE command.

If a record from a non-SQL source is not long enough to supply data to all fields mapped to target columns, each target column whose source field is missing must have a default value defined for it.

In general, if a source record from a non-SQL source does not end exactly at a field boundary, an error occurs. These exceptions apply:

- If the record ends in the middle of a VARCHAR field, the end of the record defines the end of the VARCHAR data.
- If the file is an EDIT file and the record ends in the middle of a field, SQL adds enough blanks to the end of the input record to fill the field. In such a case, blanks must be acceptable in that column of the source record. For example, a decimal field would not accept blanks; a character field would.
- If the record contains an array defined by an OCCURS DEPENDING ON clause and at least one element of the array is present, the field that contains the count must be present and the number of elements in the record must be equal to the value of the field that contains the count.
- Move options associate source fields and target fields so that data is transferred from each source field to its corresponding target field. Some considerations for using move options follow:
 - If you move data from a table to a table or between a non-SQL object and a table and you do not specify MOVE, MOVEBYNAME, or MOVEBYORDER, SQL uses MOVEBYORDER.
 - You cannot specify any move options when both source and target are non-SQL objects.

- Field formats

If a non-SQL object is the source or target, LOAD loads only those fields whose DDL descriptions conform these rules:

- The field must be elementary, unless it is the special DDL group that represents a variable-length character string, in which case the field is handled as one field during the LOAD operation. This DDL group has this structure and is converted to a column with data type VARCHAR:

```
02 A-VARCHAR
03 LEN PIC S9(4) COMP.
03 VAL PIC X(len).
```

The subfields must be named LEN and VAL, and their pictures must be exactly as shown.

- The field must not be a filler field.
- LOAD ignores Level-88 CONDITION-NAME clauses and Level-66 RENAMES clauses.
- Unless you specify a REDEFINE clause in the REDEFINE option of the LOAD command, LOAD ignores the clause and uses the original field description.

- Data type compatibility and field conversions

If you are loading data from one SQL table to another SQL table, or between an SQL table and an Enscribe file, in either direction, the data type (including the character set) of each source field or column must be compatible with the data type of the corresponding target field or column.

Character fields are compatible if the associated character sets are compatible (as described previously in this entry) and if the target field or column is large enough to hold the values loaded from the source field. If you specify TRUNCATION ON, the latter part of this requirement is always met.

If you do not specify TRUNCATION ON and a source field has a fixed-length character data type, LOAD can determine whether the source field is compatible with the target field before loading actual data. However, if a source field has a variable-length character data type, compatibility between the source and target field depends on the actual length of the specific records or rows in the source data. In this case, LOAD returns an error if it encounters an incompatible record during the actual loading.

Except for the Enscribe types COMPLEX and LOGICAL, any numeric field is compatible with any other numeric field as long as the target field is large enough to hold the values from the source field.

The column size for the INTERVAL data type can be found in the COLUMNS table using a command:

```
SELECT colname, colsize from COLUMNS WHERE TABLENAME LIKE
"%table%";
```

An interval is expressed as a multiple of the smallest unit in the INTERVAL type. For example, if the column in the table is defined as HOURS TO SECONDS, the value of the interval "1:2:3" is 3723, which is the number of seconds in 1 hour, 2 minutes, and 3 seconds.

SQL does not support the Enscribe data types COMPLEX and LOGICAL, but an Enscribe field of either one of these data types is compatible with an SQL column of data type CHARACTER with the same length as the field.

The SQL date-time and INTERVAL data types have no directly corresponding Enscribe types. (SQL date-time and INTERVAL types existed before the DDL types of the same name but are not currently equivalent for COPY and LOAD operations.)

When you load an Enscribe field into an SQL DATETIME column, the Enscribe field must have the BINARY 64 data type and must contain a Julian timestamp (a 64-bit value that contains the number of microseconds between 00:00 January 1, 4713 B.C. and the date and time it represents). SQL converts the Julian timestamp into an SQL DATETIME value and then stores the DATETIME value in the DATETIME column.

- **Implied DDL descriptions**

When you load between an SQL table and a non-SQL object (in either direction) and do not provide a DDL record for the non-SQL object, LOAD operates as if a DDL record description constructed according to these rules was supplied, and MOVEBYORDER is ON. The DDL record description contains one field for each column in the table. The type of the DDL field is determined by the type of the SQL column it corresponds to:

SQL Data Type	Enscribe Data Type
CHAR (n)	PIC X(n)
PIC X(n)	PIC X(n)
CHAR VARYING (n) or VARCHAR(n)	02 <field name> 03 LEN PIC S9(4) COMP 03 VAL PIC X(n) The value of the LEN field describes how many characters in the VAL field comprise the current value of the field.
DECIMAL(n, s)	PIC S9(n-s)V9(s). ¹
DECIMAL(n,s) UNSIGNED	PIC 9(n-s)V9(s).
NUMERIC(n,s)	BINARY 16,s. if 1 <= n and n <= 4 BINARY 32,s. if 5 <= n and n <= 9 BINARY 64,s. if 10 <= n and n <= 18

SQL Data Type	Enscribe Data Type
SMALLINT	BINARY 16.
SMALLINT UNSIGNED	BINARY 16 UNSIGNED.
INT	BINARY 32.
INT UNSIGNED	BINARY 32 UNSIGNED.
LARGEINT	BINARY 64.
PIC 9(d)V9(s)	PIC 9(d)V9(s).
PIC S9(d)V9(s)	PIC S9(d)V9(s). ¹
PIC 9(d)V9(s) COMP	same as NUMERIC(d+s,s) UNSIGNED
PIC S9(d)V9(s) COMP	same as NUMERIC(d+s,s)
FLOAT	FLOAT 64.
FLOAT(n)	FLOAT 32. if 1 <= n and n <= 22 FLOAT 64. if 23 <= n and n <= 54
DOUBLE PRECISION	FLOAT 64
REAL	FLOAT 32
DATETIME, DATE, TIME, or TIMESTAMP	BINARY 64
INTERVAL	BINARY 16. if column size = 2 BINARY 32. if column size = 4 BINARY 64. if column size = 8

¹ The Enscribe data type makes the sign a separate character, that is, the sign is not embedded.

- When both the source and target of LOAD are SQL tables, the rules for null values are the same as in any DML statement. When you perform a LOAD command on one SQL table and one non-SQL object, you must give special attention to null values, because Enscribe does not have a null value distinct from all other values of a field.

LOAD implements a convention for indicating that a field of a non-SQL object has the null value. The Enscribe DDL language defines NULL to be an attribute of data fields. The NULL attribute is a one-byte value, specified either as an integer value from 0 to 255 or as a single character in quotes. The convention that LOAD implements is that an Enscribe field has the null value if each byte of the field contains the value specified by the NULL attribute of that field.

For example, if the Enscribe DDL description of a file contains a field defined as:

```
02 F1 PIC X(10) NULL 0.
```

then the field normally contains ASCII character data. However, if the field contains binary 0 in each of the 10 bytes, LOAD would consider the field to have the null value.

If the Enscribe DDL description of a file contains a field defined as:

```
02 F1 PIC X(10) NULL "*" .
```

the field normally contains ASCII character data, but if the field contains "*" in each of the 10 bytes (that is, if the field contains the value "*****"), LOAD would consider the field to have the null value. As long as "*****" is not a value that the field could contain in normal operation of the application, then "*" is an acceptable NULL attribute for the field.

If the DDL description of a file contains a field defined as:

```
02 F1 TYPE BINARY 16 NULL 1 .
```

the field potentially can have any value from -32768 to 32767. However, if the field contains the value 257 (the value obtained when you place the value 1 in each of the two bytes of the field), LOAD would consider the field to have the null value.

When loading data from a non-SQL object to an SQL table, LOAD uses this null value convention only if the LOAD command includes the USESQLNULLS option. If the LOAD command includes the USESQLNULLS option, and if a field of a record of the non-SQL object contains the field's NULL attribute value repeated in each byte of the Enscribe field, LOAD sets the corresponding column of the row written to the SQL table to null.

If the LOAD command does not include the USESQLNULLS option, LOAD never interprets any field of the non-SQL object to have a null value (even if the Enscribe DDL record description includes a NULL attribute for the field). Therefore, even if a field of the Enscribe record contains its NULL attribute value in each byte of the field, the field's value is copied to the column corresponding to that field. It is not converted to the SQL null value.

When loading data from an SQL table to a non-SQL object, the USESQLNULLS option is irrelevant. If an Enscribe field description includes the NULL attribute, LOAD always converts an SQL null value to the Enscribe field value obtained by placing the field's NULL attribute value into each byte of the field. If an SQL column contains a null value and the Enscribe field corresponding to the column does not include the NULL attribute in the description of the field, LOAD reports file-system error 1140.

This convention for defining null values for Enscribe data sometimes is difficult to use. You can process character fields without trouble because character fields normally contain displayable ASCII characters in each byte. Therefore, choosing a value for the NULL attribute of the field that is not the code of a displayable ASCII character works easily. When you put that NULL attribute value into each byte of the field, that gives a value for the field that is not an expected normal value for the field. Often the field is known not to ever contain

some of the ASCII printable characters, in which case using one of those characters as the NULL attribute works correctly.

For DECIMAL fields, the same thing is true, because each byte of a DECIMAL field normally contains the ASCII code for one of the characters "0" through "9", so it is easy to choose a value for the NULL attribute that will not yield a null value that is the same as any expected value for the field.

For binary fields, it is sometimes difficult to choose a value for the NULL attribute that does not yield an expected value of the field when that NULL attribute value is placed in each byte of the field. In the example of the BINARY 16 field, above, if the NULL attribute is chosen to be 1, that gives a field value of 257 when 1 is put into both bytes of the field. If 257 is a value that the field could contain in normal operation of the application, NULL 1 is not a good choice for the NULL attribute. Often, numeric fields have a limited range of values that are expected to be in the field and that allows you to pick a NULL attribute value which yields an out-of-range value for the field when the NULL attribute value is placed in each byte of the field. For example, if the normal range of values for a BINARY 16 field is 0 through 32767, a NULL attribute value of 255 is a good choice, because putting 255 into both bytes of the field gives a value of -1 for the field, which is not one of the normal values expected for the field.

If a binary field has no restriction on the values it might contain, this convention for defining a null value cannot be used for that field.

To avoid these limitations for specifying the NULL attribute value, use the SQLNULLABLE attribute supported by DDL. Support for this attribute is enabled in LOAD only if the SQLNULLABLE option is specified for the LOAD command. Otherwise, LOAD ignores this attribute. Use this option only for SQL to Enscribe and Enscribe to SQL load operations.

For SQL to Enscribe load operations, LOAD adds null indicators for all the Enscribe fields that have the SQLNULLABLE attribute set in the DDL definition. For LOAD operations without the TARGETREC option, the null indicators are added for all the Enscribe fields having the corresponding SQL column as nullable.

For Enscribe to SQL load operations, LOAD reads null indicators for all the Enscribe fields that have the SQLNULLABLE attribute set in the DDL definition. For LOAD operations without the SOURCEREC option, the null indicators are read for all the Enscribe fields having the corresponding SQL column as nullable.

In all these cases, the null indicator is of two bytes with all bits set to one to indicate a null value, and all bits set to zero to indicate a nonnull value.

- These rules govern the transfer of data across character sets. A LOAD that violates these rules terminates with an error.

Source and Target File Types	Source Field Character Set	Target Field Character Set
SQL to SQL	UNKNOWN	Any character set
	ISO88591	ISO88591

	ISO88599	ISO88599
	KANJI	KANJI
	KSC5601	KSC5601
	SQL to Enscribe	UNKNOWN
	ISO88591	PIC X or PIC N

SQL to Enscribe	ISO88599	PIC X
	KANJI	PIC N
	KSC5601	PIC N
	Enscribe to SQL	PIC X
Enscribe to SQL	PIC N	Any character set
	PIC N	Any character set

For example, if the source field character set is UNKNOWN, you can copy it to a target field associated with any character set. If the source field character set is one of the nine supported ISO character sets, you can copy it only to a target field associated with that same character set.

In addition, if you load double-byte data into a single-byte field or load single-byte data into a double-byte field, the target field must be the same length, in bytes, as the source field.

(Enscribe-to-Enscribe loads do no field-by-field conversion, so that case is not shown in the previous table.)

- The FIRST KEY option specifies the starting point in the input table or file for a LOAD operation. Unlike the FIRST KEY clause in the CREATE TABLE and ALTER TABLE statements, FIRST KEY does not interpret the key value or values in terms of the key's data type, nor does it match them with specific columns. Instead, it accepts a series of data values and matches them byte-for-byte against the key.

Therefore, you must make sure the value or values you supply represent the desired data in the actual key in internal SQL format. Additionally, you should supply a complete value for each column. You can specify only a portion of the last column in the FIRST KEY specification, but SQL uses those bytes as the leftmost bytes, which might produce an unintended result.

Guidelines are:

- You can use the FIRST KEY *value* option for every type of file or table except key-sequenced files or tables.
- You should use the FIRST KEY (*value*, *value* ...) option only for key-sequenced files or tables.
- You can specify string literals or numeric literals. A string literal occupies as many bytes as there are characters in the literal. Each numeric literal occupies one byte.
- To specify a value for a CHAR column, enclose the value in quotes. For example:

```
FIRST KEY ("value")
```

To include nondisplayable characters, use the equivalent ASCII numeric form. This example specifies the BEL control character, with a numeric value of 7 as the fifth byte in an 8-byte key value:

```
FIRST KEY ("asdf",7,"jkl")
```

Be sure to supply enough bytes to fill the column. Pad the string with blanks or the appropriate pad character for your application.

You must express the value of FIRST KEY *value* for an Enscribe entry-sequenced file as 32 bits for block number followed by 32 bits for record number within the block.

You must express the value in FIRST KEY *value* for an entry-sequenced table as 24 bits for sector number followed by 9 bits for the record number within the block.

- To specify a value for a VARCHAR column, handle the value as a CHAR value and pad the value with blanks (or the appropriate pad character for your application) to the maximum size of the VARCHAR column.
- To specify a value for a binary column (such as SMALLINT, INTEGER, or LARGEINT), specify as many numeric literals as there are bytes in the column. For example, each SMALLINT column is two bytes long, so specify two numbers for each SMALLINT column. To determine the size of your target key value, look at the binary representation of the key value and determine the number of 8-bit groups (bytes) in the value.

One way to figure out byte values for positive values is to divide repeatedly by 256 until you reach a quotient of zero, and take the remainders in reverse order. For example, to represent a value of 1,000,000 for a four-byte INTEGER column:

```
1,000,000 divided by 256 is 3906, with a remainder of 64
    3,906 divided by 256 is   15, with a remainder of 66
        15 divided by 256 is    0, with a remainder of 15
```


To specify 1,000,000 as a four-byte INTEGER value, start with 0, followed by 15, 66, and 64:

```
FIRST KEY (0,15,66,64)
```

This example specifies a key value for a table with a four-column key of varying types: CHAR(2), SMALLINT, CHAR(1), and SMALLINT, starting at the values "ab", 20, "x" and 10:

```
FIRST KEY ("ab", 0, 20, "x", 0, 10)
```

The byte string in this example is 7 bytes long—one byte for each ASCII character and one byte for each numeric value.

- To specify a value for a numeric column (such as NUMERIC(n), NUMERIC(n,m), PIC ((n) COMP, or PIC 9(n) COMP), determine whether the value is represented as 2, 4, or 8 bytes. Adjust the number to an integer value by multiplying by one factor of 10 for each digit after the implied decimal point, then determine the number of 8-bit groups in the value. Specify each byte, separated by commas.

This example specifies a value of 10.5 for a column declared as NUMERIC(6,2) or PIC 9999V99 COMP:

```
FIRST KEY (0,0,4,26)
```

There are two digits after the implied decimal point, so multiply the value by 10**2 (or 100) to get 1050. Convert 1050 to byte values, to get 4,26.

- To specify an unsigned decimal column (such as DECIMAL(n) UNSIGNED), enter the value as a string, like a CHAR value. If the value includes a fraction, omit the decimal point but enter all digits, including leading and trailing zeros.
- To specify a negative value for a signed decimal column (such as DECIMAL(n) or PIC S9(n)), specify the value as a numeric byte value with the initial bit set and the remaining characters as ASCII. To do this, take the byte value of the first digit (for example, 0 is an ASCII 48) and add it to 128, the value obtained with the high-order bit set. Specify the remaining bytes in quotes.

This example specifies a value of -10 for a DECIMAL(4) field:

```
FIRST KEY (176,"010")
```

To specify a positive value for a signed decimal column, use the guidelines described previously for unsigned decimal columns.

- To specify a value for a DATETIME column, use numeric byte values. The year is stored in two bytes; the other parts are stored in one byte each. The fraction is stored in four bytes. SQL stores only the portion of the DATETIME field that is declared for the column.

This example specifies a value of 1993-03-01 for a DATETIME YEAR TO DAY column:

```
FIRST KEY (7,201,3,1)
```

The value 7,201 for the year 1993 is obtained by dividing by 256 and using remainders, as described in the preceding binary column discussion.

- To specify a value for an INTERVAL column, first determine the number of bytes required. Find the entry for the column in the COLUMNS catalog table. Do not use INVOKE output; this does not describe the internal representation of the column. Next, form the value as described for SMALLINT, INTEGER, or LARGEINT columns, expressed as a multiple of the smallest unit in the interval.
- To specify a value for a floating point column, you must convert the floating point value into a series of bytes. One way to do this is to write a program that redefines a floating point variable as an array of bytes. Store the desired floating point value into the variable, and then use the byte values from the array of bytes for the FIRST KEY specification.
- Using the FIRST *key-specifier* ALTKEY option

These considerations apply to the use of the FIRST *key-specifier* ALTKEY option:

- For nonunique alternate keys, the key includes the primary key after the last column of the alternate key. You do not need to include primary key values in your specification unless you want to include enough of the primary key value to distinguish a specific row.
- To specify the value of an alternate key columns that is nullable, include two bytes with the value 0 (for the null indicator), followed by the internal representation for the alternate key value as described previously in Using the FIRST KEY Option.

This example specifies an alternate key value of “KAQB” for a nullable PIC X(4) column:

```
FIRST keyspec ALTKEY (0,0,"KAQB")
```

- To specify a null alternate key value, include two bytes with the value 255 followed by as many bytes with value 0 as needed to complete the column value.

This example specifies an null value for a nullable PIC X(4) alternate key column:

```
FIRST keyspec ALTKEY (255,255,0,0,0,0)
```

- Using FIRST and PARTONLYIN options

If both FIRST and PARTONLYIN options are used together, only the FIRST option is executed. The PARTONLYIN option is ignored. To achieve results from both options in LOAD or APPEND commands, first load the source table with the PARTONLYIN option to a new table having the same schema, and then load or append the new table with the FIRST option to the original target table.

- Using SHARE and PARTONLYIN options

The SQLCI LOAD command ignores the PARTONLYIN if it is combined with the SHARE option and reads all the source partitions.

- HP encourages the use of BLOCKIN and RECIN attributes when the input file is an unstructured file. In this case, the RECIN must be set to the actual row length and BLOCKIN must be set to the largest multiple of the RECIN value between 1 through 32,767.

Example—LOAD

Suppose that the ODETAIL table on subvolume \$VOL1.SALES is empty, and you want to load it with data from a non-SQL object named OBASE that resides on subvolume \$VOL2.SALES. You also want to specify these requirements for the load operation:

- The source file is unsorted; a temporary scratch file named \$SPOOL.SCR.TEMP is to be used during the sorting phase of the operation.
- The source file is described by a DDL RECORD description named OFORMAT. This RECORD entry is located in a DDL dictionary on the \$VOL2.SALES.
- Each field in the DDL RECORD description is to be mapped to a column of the same name in the target table.

This command performs the specified LOAD:

```
>> LOAD $VOL2.SALES.OBASE,$VOL1.SALES.ODETAIL
+> SCRATCH $SPOOL.SCR.TEMP
+> SOURCEDICT $VOL2.SALES
+> SOURCEREC OFORMAT
+> MOVEBYNAME;
```

LOCK TABLE Statement

LOCK TABLE is a DCL statement that locks a whole table (or the underlying tables of a view) and its indexes, limiting other accesses to the table and its indexes while you or your program execute DML statements.

<pre>LOCK TABLE { <i>name</i> } IN { SHARE } MODE { EXCLUSIVE }</pre>

name

is the name (or an equivalent DEFINE) of a table or view to lock.

SHARE or EXCLUSIVE

specifies the locking mode:

SHARE	Others can read, but not delete, insert, or update the table or view.
EXCLUSIVE	Others can read with BROWSE access, but cannot read with STABLE or REPEATABLE access and cannot delete, insert, or update.

If you request a SHARE lock on a table locked with an EXCLUSIVE lock by another user, your request waits until the EXCLUSIVE lock is released.

If you request an EXCLUSIVE lock on a table and any part of the table is locked by another user, your request waits until the lock is released, or until your lock request times out and SQL returns an error message.

Considerations—LOCK TABLE

- LOCK TABLE requires authority to read the table. Locking a view requires authority to read the view and its underlying tables.
- A SELECT statement automatically acquires SHARE locks unless you specify BROWSE access. (DELETE, INSERT, and UPDATE automatically acquire EXCLUSIVE locks.) You can use LOCK TABLE with the EXCLUSIVE option to force use of EXCLUSIVE locks for a subsequent SELECT, but LOCK TABLE will lock the whole table.
- Follow each LOCK TABLE statement with a CONTROL TABLE TABLELOCK ON directive for the same table. This directive specifies that the table will be locked at execution time. The compiler uses this information to select the most efficient execution path for your data.
- Audited tables never need to be explicitly unlocked. An audited table can be locked only within a TMF transaction and is automatically unlocked when the transaction ends. (UNLOCK TABLE does not affect audited tables.)

You unlock nonaudited tables with UNLOCK TABLE or FREE RESOURCES. In SQLCI (but not in host programs), nonaudited tables are sometimes unlocked automatically. If the AUTOWORK session option (without AUDITONLY) is on, locked nonaudited tables are unlocked after the next DML statement. Locked tables are also unlocked when you issue COMMIT WORK or ROLLBACK WORK (without AUDITONLY) to end a user-defined transaction, when you press the Break key when BREAK_KEY is on, or when your SQLCI session ends.

- LOCK TABLE attempts to lock all partitions and indexes of any table it locks. If a partition or index is not available or if the lock request times out, LOCK TABLE displays a warning (in SQLCI) or returns the name of the unavailable partition or index to the SQLCA (in host programs) and continues to request locks on other partitions and indexes.

Examples—LOCK TABLE

- This example locks an audited table with an EXCLUSIVE lock (presumably at a time when few users need access to the database) to perform an update. The CONTROL TABLE statement ensures the most efficient operation. COMMIT WORK automatically unlocks the table when it ends the transaction.

```
>> VOLUME $VOL1.PERSNL;
>> BEGIN WORK;
>> CONTROL TABLE EMPLOYEE TABLELOCK ON;
>> LOCK TABLE EMPLOYEE IN EXCLUSIVE MODE;
--- SQL operation complete
>> UPDATE EMPLOYEE SET SALARY = SALARY * 1.05
+> WHERE JOBCODE <> 100;
--- 45 row(s) updated.
>> COMMIT WORK;
```

- This example deletes all rows of the JOB table with a job code that is not assigned to any employee. Suppose that the JOB table is nonaudited and you want locks held for several transactions. Because the EMPLOYEE table is audited and you are locking it, you must define a TMF transaction. At the end of the transaction, the EMPLOYEE table lock is released by the system. Unless AUTOWORK is set to ON without AUDITONLY, you must use the UNLOCK TABLE command to release the lock on the JOB table because the table is nonaudited.

```
>> VOLUME $VOL1.PERSNL;
>> BEGIN WORK;
>> CONTROL TABLE JOB TABLELOCK ON;
>> CONTROL TABLE EMPLOYEE TABLELOCK ON;
>> LOCK TABLE JOB IN EXCLUSIVE MODE;
--- SQL operation complete
>> LOCK TABLE EMPLOYEE IN SHARE MODE;
--- SQL operation complete
>> DELETE FROM JOB WHERE JOBCODE NOT IN (SELECT DISTINCT
+> JOBCODE FROM EMPLOYEE);
--- 8 row(s) deleted.
>> COMMIT WORK AUDITONLY;
...
>> UNLOCK TABLE JOB;
--- SQL operation complete
```

- This example locks a nonaudited table (SALES.PARTS) and then explicitly unlocks it after processing:

```
EXEC SQL
  LOCK TABLE SALES.PARTS IN EXCLUSIVE MODE;
EXEC SQL
  CONTROL TABLE SALES.PARTS TABLELOCK ON;
...
EXEC SQL
  UNLOCK TABLE SALES.PARTS;
EXEC SQL
  CONTROL TABLE SALES.PARTS TABLELOCK ENABLE;
```

Locking

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

Default locking normally protects data but reduces concurrency. If your application has problems with lock contention (as indicated by MEASURE LOCKWAIT counts or by the value in the SQLSA WAITS field), you might want to use options that control the characteristics of locks.

Locks have these characteristics:

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

Lock Duration

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

You choose lock duration by specifying access options: STABLE for a short time or REPEATABLE for a long time. For more information, see [Access Options](#) on page A-1.

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

[Table L-1](#) on page L-49 lists SQL operations that release locks and shows the effects of STABLE and REPEATABLE access mode on lock duration. The table assumes default locking (shared locks for reads and exclusive locks for updates) and also assumes the locking strategy uses the minimum number of locks. SQL can use additional locks (either row locks or table locks) if it determines that additional locks are necessary to protect data integrity or to provide faster data access.

If a partitioned table is randomly accessed with STABLE ACCESS, the lock is held until the next access to the same partition (not the next access to the table). This can cause a lock to be held longer than expected.

When using sequential access with Sequential Block Buffering and STABLE access, SQL holds the lock until it attempts to access a row that is not in the buffer. Thus, rows at the beginning of the buffer remain locked while subsequent rows are being fetched by the program.

Table L-1. Lock Release Summary

Operation	Audited Tables/Views		Nonaudited Tables/Views	
	STABLE	REPEATABLE	STABLE	REPEATABLE
SELECT INTO	RA	KA	RA	KA
FETCH cursor updateable ¹	RS,KE ²	KA	RA	KA
FETCH cursor not updateable	RS ³	KA	RA	KA
Set UPDATE or DELETE	RS,KE	KA	RA	KA
INSERT	KA	KA	RA	KA
CLOSE cursor	RS,KE	KA	RA	KA
FREE RESOURCES	RS,KE	KA	RA	RA
FREE RESOURCES AUDIT ONLY	RS,KE	KA	N.A.	N.A.
UNLOCK TABLE	N.A.	N.A.	RA	RA
TMF transaction abort/commit	RA	RA	RA	RA
COMMIT or ROLLBACK WORK AUDITONLY	RA	RA	N.A.	N.A.

A Any type of lock R Release locks

E Exclusive locks S Shared locks

K Keep locks

Example: RS,KE is release shared locks; keep exclusive.

¹ Locks for UPDATE and DELETE through a cursor are processed through FETCH operations.

² The lock is kept if the record is updated.

³ The lock is kept if the record is deleted.

Lock Granularity

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

The LOCKLENGTH file attribute for a table or index controls the granularity of locks for the table or index. You can control locks for an entire table and its associated indexes with LOCK TABLE and CONTROL TABLE; otherwise, SQL determines the granularity by considering the access option you specify, the table size and definition, and the estimated percentage of rows the query will access.

SQL can automatically increase the granularity of locks for a particular transaction, depending on processing requirements. This is called lock escalation. For example, if a process holds many row locks in different partitions of a partitioned table, SQL might escalate the row locks to a table lock. If a process holds many row locks on the same

partition of a partitioned table, SQL might escalate the row locks to a partition lock. A partition lock applies only to the specific partition and not to the entire table. For a nonpartitioned table, a partition lock is a table lock. If you do not want lock escalation, use the TABLELOCK OFF option on the CONTROL TABLE statement.

Lock Mode

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

SHARE lock mode allows multiple users to lock and read the same data. EXCLUSIVE lock mode limits access to locked data to the lock holder and to other users who specify BROWSE (but not STABLE or REPEATABLE) access. Lock modes are the same whether you choose STABLE or REPEATABLE access.

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

- For queries only, you can choose BROWSE access.
- On the LOCK TABLE statement, you can choose either EXCLUSIVE or SHARE.
- On the SELECT statement, you can specify IN EXCLUSIVE MODE or IN SHARE MODE.

Lock Holder

The lock holder of an object depends on whether the object is audited or non-audited as follows:

- Locks on audited objects are held by the TMF transaction in which the request to access the data was made. The Transaction ID of the TMF Transaction that locked the object is the lock holder.
- Locks on nonaudited objects are held by the process that opens the object: either SQLCI or a host program.

Process ID or Process name of the process (host program/SQLCI) which has locked the object, is the lock holder.

The lock holder can release a lock only under the following scenarios:

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

An AUDITONLY option is available for the SQLCI AUTOWORK session command and for these SQL statements:

COMMIT WORK

FREE RESOURCES

ROLLBACK WORK

If you specify **AUDITONLY**, the process releases locks on audited objects only. You can use this option if you want to hold locks on nonaudited objects throughout a series of transactions.

Stopping or abnormal termination of a process frees any locks the process holds on nonaudited tables. **CLOSE** and **FREE RESOURCES** release locks on nonaudited tables.

LOCKLENGTH File Attribute

LOCKLENGTH is a Guardian file attribute that specifies the number of leading bytes used to identify rows for generic locking. **LOCKLENGTH** applies to key-sequenced tables and indexes.

<code>LOCKLENGTH num-bytes</code>

num-bytes

is an integer between 0 and the number of bytes in the key that specifies the number of leading bytes in the primary or clustering key (including **SYSKEY** if it exists) to use to identify rows for generic locking.

LOCKLENGTH 0 (the default) indicates that the entire length of the primary or clustering key should be used.

Consideration—LOCKLENGTH

Increasing **LOCKLENGTH** increases lock granularity, reducing the number of locks issued and the amount of lock escalation. Reducing the number of locks improves performance but reduces concurrency, because one lock controls a larger group of rows.

Example—LOCKLENGTH

This example creates a table with a **LOCKLENGTH** that fits the first column of a two-column key. Each row represents part of an order, and the first column is the order number. When order-processing applications access the table, SQL issues a single lock against all rows for an order. (The default **LOCKLENGTH** would cause a separate lock for each row in the order.)

```
CREATE TABLE SALES.ODETAIL (
  ORDERNUM    NUMERIC (6) UNSIGNED,    -Column length
  PARTNUM     NUMERIC (4) UNSIGNED,      6 bytes
  QTY_ORDERED NUMERIC (5),
  PRIMARY KEY ( ORDERNUM , PARTNUM ) -Key declaration
```

)
LOCKLENGTH 6;

—Lock length 6 bytes

LOG Command

LOG is an SQLCI command that starts or stops logging to a file. SQLCI logs the commands you enter and the information the commands display. You can also use an option to log only the commands. This option allows you to create an SQLCI OBEY command file from the log file directly. SQLCI does not log the FUP, EDIT, PERUSE, and TEDIT commands.

```
LOG [ log-file [COMMAND[S] ] [ CLEAR ] ] ;
```

log-file

identifies the current log file and starts logging. For *log-file*, specify a device, process, or disk file. You cannot log to any current output file (such as the INVOKE TO, OUT, or OUT_REPORT file) or to any open file other than a terminal or process. If you specify a nonexistent disk file, an EDIT file is created. When you enter the LOG command, the previous log file is closed.

To stop logging and close the log file, omit *log-file*.

COMMANDS

allows SQLCI to log the commands only.

CLEAR

clears the new log file of all existing data before logging begins. If you omit this option, logging information is appended to the file. CLEAR is ignored unless the file is a disk or process file.

This information is written to the log file:

- All text that SQLCI displays or prints, including output from commands such as SHOW, data from SELECT commands, and diagnostic messages.
- All lines you type (preceded by the current prompt).

The final version of a command is written to the log file without the extra characters you enter while making changes with FC.

Example—LOG

This example starts logging SQLCI output to the file SUBV2.MAYLOG:

```
>> LOG SUBV2.MAYLOG;
```

LOGICAL_FOLDING Option

LOGICAL_FOLDING is an option of the SQLCI report writer SET LAYOUT command that specifies where to break a default detail line (one for which there is no DETAIL command) that does not fit within the report width.

```
LOGICAL_FOLDING { ON }
                 { OFF }
```

ON

breaks the line before the first print item that does not fit. ON is the default.

OFF

breaks the line exactly at the right margin, even if the break is in the middle of a print item.

Considerations—LOGICAL_FOLDING

- The default right margin is the width of the current output device (OUT_REPORT or OUT file). The default left margin is 0. Report width is right margin minus left margin.
- The LOGICAL_FOLDING option affects only lines that are folded because of the report width. LOGICAL_FOLDING does not affect print lines that wrap because of the output device length; such folding is controlled by the WRAP option.
- LOGICAL_FOLDING does not affect output lines other than default detail lines. For example, report writer displays output from the REPORT TITLE, BREAK TITLE, and REPORT FOOTING commands according to the command specifications, regardless of the setting of the LOGICAL_FOLDING option.

Example—LOGICAL_FOLDING

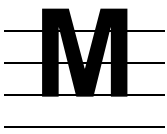
This example prints a report with LOGICAL_FOLDING ON (by default), and then with LOGICAL_FOLDING OFF (after SET LAYOUT):

```
>> SET LAYOUT RIGHT_MARGIN 60;
>> SET LIST_COUNT 2;
>> SELECT * FROM INVENT.SUPPLIER;
```

SUPPNUM	SUPPNAME	STREET
CITY	STATE	POSTCODE
1	NEW COMPUTERS INC	1800 KING ST.
SAN FRANCISCO	CALIFORNIA	94112
2	DATA TERMINAL INC	2000 BAKER STREET
LAS VEGAS	NEVADA	66134

```
S> SET LAYOUT LOGICAL_FOLDING OFF;
S> LIST FIRST 2;
```

SUPPNUM	SUPPNAME	STREET	CI
-----	-----	-----	--
TY	STATE	POSTCODE	
-----	-----	-----	
1	NEW COMPUTERS INC	1800 KING ST.	SA
N FRANCISCO	CALIFORNIA	94112	
2	DATA TERMINAL INC	2000 BAKER STREET	LA
S VEGAS	NEVADA	66134	



MAX Function

MAX is a function that determines the maximum value within a set of values. The type of the result depends on the type of the argument.

```
MAX { ( [ ALL ] expression ) }  
    { ( DISTINCT column      ) }
```

[ALL] expression

specifies an expression that indicates the set of values from which to determine a maximum value.

The expression must include a value from each row of the result table (that is, at least one column from the result table), and cannot include the COUNT, AVG, MIN, or SUM functions, or another MAX function. For example,

```
MAX ( SALARY )  
MAX ( PARTCOST * QTY_ORDERED )
```

ALL is an optional keyword that does not change the meaning of the clause. SQL uses all rows (whether or not you specify ALL) unless you use the DISTINCT clause, described next.

DISTINCT column

specifies a set of distinct column values from each row of the result table to determine a maximum value. The column cannot be a column from a view that corresponds to an expression in the view definition.

If you specify DISTINCT in more than one MAX function in the same statement, the functions must reference the same column.

Duplicate rows are eliminated only if you specify DISTINCT; otherwise, all rows are included, whether or not you specify ALL.

Specifying DISTINCT with the MAX function does not restrict the use of DISTINCT with AVG, SUM, MIN, or COUNT.

Considerations—MAX

- If you specify an expression or column with a character data type as the argument to MAX, the collation used for the comparison is the collation associated with the argument. SQL uses a binary comparison if no collation is associated with the argument.

- MAX is evaluated after eliminating all null values from the aggregate set. If the result set is empty, MAX returns a null value.
- A host variable that receives the result of the MAX function must have an indicator variable to handle a possible null value. (For more information about using indicator variables, see the SQL/MP programming manual for your host language.)

Example—MAX

To display the maximum value in the SALARY column, type:

```
>>SELECT MAX (SALARY) FROM PERSNL.EMPLOYEE;
(EXPR)
-----
      175500.00
--- 1 row(s) selected.
```

MAXEXTENTS File Attribute

MAXEXTENTS is a file attribute that specifies the maximum number of extents that can be allocated for an unpartitioned file or for each partition of a partitioned file. MAXEXTENTS applies to key-sequenced, relative, and entry-sequenced tables and to indexes.

MAXEXTENTS <i>num-extents</i>

num-extents

is an integer from 1 to 959 (but not less than the number of extents currently allocated for the file) that specifies the maximum number of extents that can be allocated. Format 1 partitions allow MAXEXTENTS up to 944. Format 2 partitions allow MAXEXTENTS up to 919. These limits are enforced by DP2 when you attempt to allocate the 945th (Format 1) or 920th (Format 2) extent.

The default is MAXEXTENTS 160.

SQL catalog manager enforces the following limits when the define `=_SQL_LATEST_MAXEXTENTS` is added:

- For FORMAT 1 partitions (whether primary or secondary), maximum allowed MAXEXTENTS limit is 944.
- For FORMAT 2 partitions (whether primary or secondary), maximum allowed MAXEXTENTS limit is 919.

Considerations—MAXEXTENTS

- You can alter MAXEXTENTS for any partition of an index or a key-sequenced table but only for the last partition of a relative or entry-sequenced table. (For tables in ascending order, the last partition is the one with the highest range of FIRST KEY

values; for tables in descending order, the last partition is the one with the lowest range of FIRST KEY values.)

- It is generally not efficient to have partitions with hundreds of extents, so you should keep MAXEXTENTS well below the allowed maximum value. If necessary, increase the number of partitions.

In addition, the maximum value for MAXEXTENTS might be lower in future releases. If that occurs, existing files with higher MAXEXTENTS values will still be valid, but those files will not be able to add additional extents beyond the current maximum value of MAXEXTENTS.

- During certain DDL operations, such as CREATE INDEX requests and ALTER TABLE or ALTER INDEX one-way move partition requests, SQL changes the value of MAXEXTENTS during the DDL operation and then, at the end of the operation, resets MAXEXTENTS to the user-specified value or the actual extents allocated, whichever is larger. If the new value exceeds the user-specified value, SQL reports the new value in a warning message at the end of the DDL operation.

Message File

The SQL message file is a key-sequenced file that contains error messages, warning messages, and help text for NonStop SQL/MP. The default message file, \$SYSTEM.SYSTEM.SQLMSG, contains messages and help text in U.S. English.

For information about specifying an alternate message file, see [SQL MSG node DEFINE](#) on page Z-18.

MIN Function

MIN is a function that returns the minimum value within a set of values. The type of the result is the type of the argument.

<pre>MIN { ([ALL] expression) } { (DISTINCT column) }</pre>
--

[ALL] *expression*

specifies an expression that indicates the set of values from which to determine a minimum value.

The expression must include a value from each row of the result table (that is, at least one column from the result table), and cannot include the COUNT, AVG, MAX, or SUM functions, or another MIN function. For example,

```
MIN (SALARY)
MIN (PARTCOST * QTY_ORDERED)
```

ALL is an optional keyword that does not change the meaning of the clause. SQL uses all rows (whether or not you specify ALL) unless you use the DISTINCT clause, described next.

`DISTINCT column`

specifies a set of distinct column values from each row of the result table to determine a minimum value. The column cannot be a column from a view that corresponds to an expression in the view definition.

If you specify DISTINCT in more than one MIN function in the same statement, the functions must reference the same column.

Duplicate rows are eliminated only if you specify DISTINCT; otherwise, all rows are included, whether or not you specify ALL.

Specifying DISTINCT with the MIN function places no restrictions on the use of DISTINCT with AVG, SUM, MAX, or COUNT.

Considerations—MIN

- If you specify an expression or column with a character data type as the argument to MIN, the collation used for the comparison is the collation associated with the argument. SQL uses a binary comparison if no collation is associated with the argument.
- MIN is evaluated after eliminating all null values from the aggregate set. If the result set is empty, MIN returns a null value.
- A host variable that receives the result of the MIN function must have an indicator variable to handle a possible null value. (For more information about using indicator variables, see the SQL/MP programming manual for your host language.)

Example—MIN

To determine the minimum value in the SALARY column, type:

```
>>SELECT MIN (SALARY) FROM PERSNL.EMPLOYEE;
(EXPR)
-----
    12000.00
--- 1 row(s) selected.
```

MODIFY CATALOG

The MODIFY CATALOG command has two options that can be specified as the *replace-spec*:

- [MODIFY CATALOG with REPLACE NODENAME](#)
- [MODIFY CATALOG with REPLACE VOLUME](#)

MODIFY CATALOG with REPLACE NODENAME

The MODIFY CATALOG with REPLACE NODENAME command modifies node names in SQL/MP catalogs on the local node. A catalog can be a user-defined catalog or the system catalog.

This command is intended for use when physically moving a disk from one node in your network to another node, or when system loading a node with a new node name or number. When this happens, changes are not automatically reflected in the catalogs and file labels. The internal consistency of the database is lost; catalogs (and objects they describe) cannot be accessed. The MODIFY commands (LABEL, CATALOG, and REGISTER) let you change the database to reflect the new information.

Before requesting a MODIFY CATALOG with REPLACE NODENAME operation for a catalog, check that the file labels of the catalog are accessible. If necessary, perform a MODIFY LABEL with REPLACE NODENUMBER operation on all catalog tables in the catalog first, if necessary.

```
MODIFY [DICTIONARY] CATALOG target-spec replace-spec

    WITH node-name [ [,] option ] ... ;

target-spec is:

    catalog-list-1 [ EXCLUDE catalog-list-2 ]

replace-spec is:

REPLACE NODENAME node-name [ ( volumeset [, volumeset]
    [EXCLUDE volumeset [, volumeset] ] ) ]

option is:

    [ | ALLOWERRORS [ OFF | ON | number-of-errors ] | ]
    [ | [ NO ] LISTALL | ]
    [ | DETAIL [ MATCH | ALL ] REPORT [ TO EMS-Collector ] | ]
    [ | [ ON | ] | ]
    [ | [ OFF | ] | ]
    [ | CHECKONLY | ]

catalog-list-n (where n = 1,2) is:

    { catalogset }
    { ( catalogset [ , catalogset ] ... ) }
```

catalog-list-1 [EXCLUDE *catalog-list-2*]

identifies one or more SQL catalogs to modify. The catalogs need not be registered in the system catalog at the time the MODIFY CATALOG with REPLACE NODENAME command is run.

To specify a single catalog, enter the name of the catalog (the name of the subvolume that contains the catalog). To specify multiple catalogs in a *catalogset*, use wild-card characters. You can use these wild-card characters:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates any name is acceptable. To specify all catalogs, use either *\$*.** or **.**
- ? Matches any single character

For example, *\$DATA.** specifies all catalogs on the volume *\$DATA*, and **.** specifies all catalogs on the node. **VOL** matches *NEWVOL*, *OLDVOL1*, and *VOL45*. *VOL?* matches *VOL1* and *VOLX*, but not *VOL* or *VOL48*.

The **MODIFY CATALOG with REPLACE NODENAME** command assumes that a subvolume contains a valid catalog if the subvolume contains the catalog table *TABLES* (file code must be 581). The optional **EXCLUDE *catalog-list-2*** clause specifies catalogs to be excluded from *catalog-list-1*.

```
REPLACE NODENAME node-name [ ( volumeset [,volumeset]
[EXCLUDE volumeset [,volumeset] ] ) ]
```

specifies the node name to replace in the catalog tables. SQL catalog tables contain file names, and those file names contain node names. Node names are changed based on the *volumeset* list. Each node name is replaced by the node name specified in the **WITH *node-name*** clause.

If the first *volumeset* list is specified, SQL replaces the node name only if it matches the node name specified by *node-name* and if the volume name in the file name matches one of the volume names specified in the *volumeset* list. The optional **EXCLUDE *volumeset*** list clause specifies volumes to be excluded from the first *volumeset* list.

node-name must be preceded by a backslash character and must consist of from one to seven alphanumeric characters. The first character must be alphabetic. You can specify either uppercase or lowercase alphabetic characters; alphabetic characters are upshifted before the comparison and substitution process.

The first *volumeset* list specifies a volume or a set of volumes. To specify a single volume, enter the name of the volume. To specify multiple volumes, use wild-card characters. You can use these wild-card characters:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates any name is acceptable. To specify all volumes, use either *\$*.** or **.**
- ? Matches any single character

For example, *\$DAT** specifies all volumes that begin with *\$DAT*. **VOL** matches *NEWVOL*, *OLDVOL1*, and *VOL45*. *VOL?* matches *VOL1* and *VOLX*, but not *VOL* or *VOL48*.

The maximum number of *volumesets* that can be specified in this clause is 10 for each *volume* list. For example, this clause is invalid because it has 11 volumes in the first *volume* list:

```
( $A* , $B* , $C* , $D* , $E* , $F* , $G* , $H* , $I* , $J* , $K* EXCLUDE $Z* )
```

If this clause is not specified, only the node name is considered.

WITH *node-name*

specifies the node name to replace occurrences of the node name specified in *replace-spec*.

node-name must be preceded by a backslash character and consist of from one to seven alphanumeric characters. The first character must be alphabetic. Either uppercase or lowercase alphabetic characters can be specified; alphabetic characters are upshifted before the comparison and substitution process.

ALLOWERRORS [OFF | ON | *number-of-errors*]

determines handling of nonfatal errors. MODIFY reports two classes of errors: fatal errors and nonfatal errors. The MODIFY CATALOG with REPLACE NODENAME command terminates immediately after reporting a fatal error. Nonfatal errors are handled depending on the value of the ALLOWERRORS option:

OFF	The MODIFY CATALOG with REPLACE NODENAME command terminates immediately after the first nonfatal error is encountered.
ON	THE MODIFY CATALOG with REPLACE NODENAME command continues, regardless of the number of nonfatal errors encountered.
<i>number-of-errors</i>	The MODIFY CATALOG with REPLACE NODENAME command continues until the number of nonfatal errors exceeds value of <i>number-of-errors</i> .

If you specify ALLOWERRORS without ON, OFF, or the number of errors, ALLOWERRORS ON is the default.

If you do not specify the ALLOWERRORS option, ALLOWERRORS OFF is the default.

When MODIFY CATALOG with REPLACE NODENAME continues processing after a nonfatal error has occurred, it advances to the next SQL catalog.

[NO] LISTALL

specifies how much detail MODIFY CATALOG with REPLACE NODENAME writes to the current OUT file. If LISTALL is specified, MODIFY CATALOG with REPLACE NODENAME reports the name of each SQL catalog considered for modification and whether or not the catalog was modified. If NO LISTALL is specified, only summary is reported.

LISTALL is the default.

```
DETAIL [ MATCH | ALL ] REPORT [ TO EMS-Collector ]
                                     [ ON ]
                                     [ OFF ]
```

specifies that detail about the MODIFY CATALOG with REPLACE NODENAME operation is to be sent in event messages to a valid EMS collector.

If MATCH is specified, detailed is reported about SQL catalog tables specified in *target-spec* that contain node names that match the criteria specified in the REPLACE clause. If ALL is specified, detailed is reported about all SQL catalog tables specified in *target-spec*.

MATCH is the default.

For information on report options, see [REPORT Option](#) on page R-3.

CHECKONLY

specifies that the catalog tables specified in *target-spec* should be checked to see if they contain node names that match the criteria specified in the REPLACE clause. No catalog tables are changed. The CHECKONLY option enables you to estimate the effect of running the MODIFY CATALOG with REPLACE NODENAME command before actually modifying the catalog tables.

Considerations—MODIFY CATALOG with REPLACE NODENAME

- You must be logged on as the super ID to run a MODIFY CATALOG with REPLACE NODENAME command unless you specify the CHECKONLY option. If you specify the CHECKONLY option, you must have authority to read the catalogs.
- SQL uses the TMF subsystem to protect the integrity of the database during the MODIFY operation. MODIFY CATALOG with REPLACE NODENAME commands are not allowed inside a user-defined transaction.

One system-defined transaction is used for each set of catalog tables modified. If an error occurs while MODIFY CATALOG with REPLACE NODENAME is modifying a catalog table, the changes made to that catalog table and all other tables in the same catalog are backed out. However, changes that have been made to other catalogs during the same instance of the MODIFY CATALOG with REPLACE NODENAME command are not backed out. Thus, for one set of catalog tables, either the node name is changed in all the tables in the set or it is not changed in any of the tables in the set.

- The MODIFY CATALOG with REPLACE NODENAME Command is one of a set of commands that uses the MODIFY DICTIONARY utility. The other related commands are [MODIFY CATALOG with REPLACE VOLUME](#) (to replace volume name in catalog), [MODIFY LABEL with REPLACE NODENUMBER](#) (to replace node number in file labels), [MODIFY LABEL with REPLACE VOLUME](#) (to replace

volume name in file label), and MODIFY REGISTER Command (to register user-defined catalogs in the local system catalog).

- Multiple MODIFY commands (including LABEL, CATALOG, and REGISTER commands) can be run concurrently on the same node as long as each command is processing a different set of files or catalogs. For example, if the node number is changed on a node that has an SQL database spread out over five volumes, five MODIFY LABEL commands can be started concurrently, each specifying a different volume to be modified. Note that in such a case, because the node name did not change, it would not be necessary to run any MODIFY CATALOG with REPLACE NODENAME commands.
- Before requesting a MODIFY CATALOG with REPLACE NODENAME operation for a catalog, do a MODIFY LABEL with REPLACE NODENUMBER operation on all catalog tables in the catalog.
- MODIFY CATALOG with REPLACE NODENAME locks one catalog table at a time. The file labels of the catalog tables are not locked. Do not request DDL or update operations until the node names are modified, including operations on partitions and on dependent objects on remote nodes.
- The MODIFY DICTIONARY utility does not handle:
 - Remote nodes. If you specify a remote file name or catalog name, the MODIFY command reports a nonfatal error. Therefore, each node with dependent objects or partitioned objects must have a version of NonStop SQL/MP that supports MODIFY CATALOG with REPLACE NODENAME commands.
 - User-defined SQL object files. For example, MODIFY CATALOG with REPLACE NODENAME does not modify a node name stored in a column of a user-defined table.
 - Names stored in SQL object program files. SQL object programs can refer to an SQL object by using either a DEFINE or the Guardian name of the object. If DEFINES are used, both the DEFINE name and the associated Guardian name of the SQL object are stored in the object program file—not in the file label. If Guardian names are used, the Guardian names are stored in the object program file in internal network form.
 - Node names in Enscribe files. If a disk containing an alternate key file or a partition file is moved to a different node, the FUP ALTER command can be used to change the Enscribe file label that references the file that moved. Note that changes must be made to the Enscribe file labels that point to the disk that moved, not to the alternate key file label or partition file label that resides on the disk that moved.
 - Dependent objects. The MODIFY CATALOG with REPLACE NODENAME command does not modify information about dependent objects that reside in other catalogs unless the other catalogs are specified.

It is the responsibility of the user to know how the database is distributed. Document the MODIFY commands that need to be run, the nodes they need to

be run on, before they are used. Prepare scripts that run the necessary MODIFY commands. When you add a new dependent object to the database, update the scripts.

While the node is in a consistent state, you can use the DISPLAY USE OF command to locate dependent objects. After MODIFY commands are run, use the VERIFY utility to verify that the database is in a consistent state.

- Partitioned objects. For a partitioned SQL object, each volume that contains a partition of the object must be specified separately. MODIFY CATALOG with REPLACE NODENAME does not automatically modify information about all partitions of a partitioned object.

It is the responsibility of the user to know how the database is partitioned. While the node is in a consistent state, issue a SELECT from the PARTNS partitions table to locate other partitions. Prepare scripts that run the necessary MODIFY commands. After running the MODIFY commands, use the VERIFY utility to verify that the database is in a consistent state.

- MODIFY CATALOG with REPLACE NODENAME does not mark SQL object programs as invalid in either the catalog or in the object program file label.
- MODIFY CATALOG with REPLACE NODENAME does not change the redefinition timestamp in either the catalog or the file label.
- You can use DEFINE names in programs to specify names of catalogs, tables, views, indexes, partitions, and other programs. The current DEFINE set at the time the program is SQL compiled is saved in the object program file. If objects or object programs specified by the DEFINES are moved between the time that the program is SQL compiled and the time that the program is run, the DEFINES must be changed to reflect the new location of the objects and object programs. This guideline is true regardless of how the database was moved and regardless of whether the MODIFY DICTIONARY utility was used to modify the node names and numbers.

The MODIFY DICTIONARY utility does not modify node names in the DEFINE set stored in the object program file. After the DEFINES are changed by the user, if automatic recompilation is enabled, the programs are automatically recompiled using the new DEFINES.

- The MODIFY DICTIONARY utility is not intended to correct the situation where a disk containing objects is moved to a new node, but the disk containing the associated catalog is not moved to the new node.
- The MODIFY CATALOG with REPLACE NODENAME command returns a nonfatal versioning error if an SQL catalog has a version newer than the version of MODIFY that is accessing it, or if the catalog has a version newer than the version of SQL/MP system software on the node where the catalog resides.

Examples—MODIFY CATALOG with REPLACE NODENAME

These examples illustrate the use of the REPLACE clause.

- In this example, the node name \TESS is replaced with the node name \FOXII if the first three characters of the volume name are \$DA:

```
REPLACE NODENAME \TESS ($DA*) WITH \FOXII
```

- In this REPLACE clause, the node name \TESS is replaced with the node name \FOXII if the volume name is either \$SAM or \$CAT or if the first three characters of the volume name are \$DA:

```
REPLACE NODENAME \TESS ($SAM,$CAT,$DA*) WITH \FOXII
```

- In this REPLACE clause, the node name \TESS is replaced with the node name \FOXII if the volume name is anything other than \$SYSTEM:

```
REPLACE NODENAME \TESS ($* EXCLUDE $SYSTEM) WITH \FOXII
```

- When you use the CHECKONLY option, note that the amount of information written to the current OUT file depends on whether LISTALL or NO LISTALL is specified. The MODIFY CATALOG with REPLACE NODENAME command produces this display for each catalog when requested with the LISTALL option:

```
Checking catalog \SYS.$VOL.CAT1.
--- \SYS.$VOL.CAT1 was modified.
Checking catalog \SYS.$VOL.CAT2.
--- \SYS.$VOL.CAT2 was not modified.
```

This summary information is included whether you specify LISTALL or NO LISTALL:

```
Summary Information:
    nnn catalog(s) require modification.
    nnn catalog(s) do not require modification.
```

MODIFY CATALOG with REPLACE VOLUME

Note. The MODIFY CATALOG with REPLACE VOLUME command is supported on systems running J06.04 and later J-series RVUs and H06.15 and later H-series RVUs only.

The MODIFY CATALOG with REPLACE VOLUME command modifies volume names in SQL/MP catalogs on the local node. A catalog can be a user-defined catalog or the system catalog. This command is intended for use when physically moving a disk from one node in your network to another node which has a similar disk volume name, or when system loading a disk with a new disk volume name. When this happens, changes are not automatically reflected in the catalogs and file labels. The internal consistency of the database is lost; catalogs (and objects they describe) cannot be accessed. The MODIFY commands (LABEL, CATALOG) enable you change the database to reflect the new information.

Before requesting a MODIFY CATALOG with REPLACE VOLUME operation for a catalog, check that the file labels of the catalog are accessible. If necessary, perform a

MODIFY LABEL with REPLACE VOLUME operation on all catalog tables in the catalog first. For more information about MODIFY CATALOG with REPLACE VOLUME, see [Considerations-MODIFY CATALOG with REPLACE VOLUME](#) for more details.

```
MODIFY [DICTIONARY] CATALOG target-spec replace-spec
WITH volume-name [ [,] option ] ... ;

target-spec is:

catalog-list-1 [ EXCLUDE catalog-list-2 ]

replace-spec is:

REPLACE VOLUME volume-name [ ( node-name1 [,node-name2]
[EXCLUDE node-name3 [,node-name4]... ] ) ]

option is:
[ | ALLOWERRORS [ OFF | ON | number-of-errors ] | ]
[ | [ NO ] LISTALL | ]
[ | DETAIL [ MATCH | ALL ] REPORT [ TO EMS-Collector ] | ]
[ | [ ON ] | ]
[ | [ OFF ] | ]
[ | CHECKONLY | ]

catalog-list-n (where n = 1,2) is:
{ catalogset }
{ ( catalogset [ , catalogset ] ... ) }
```

catalog-list-1 [EXCLUDE *catalog-list-2*]

identifies one or more SQL catalogs to modify. The catalogs need not be registered in the system catalog at the time the MODIFY CATALOG with REPLACE VOLUME command is run. To specify a single catalog, enter the name of the catalog (the name of the subvolume that contains the catalog). To specify multiple catalogs in a catalogset, use wild-card characters. You can use these wild-card characters:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates that any name is acceptable. To specify all volumes, use either \$*.* or *.*.
- ? Matches any single character

For example, \$DATA.* specifies all catalogs on the volume \$DATA, and *.* specifies all catalogs on the node. *VOL* matches NEWVOL, OLDDVOL1, and VOL45. VOL? matches VOL1 and VOLX, but not VOL or VOL48.

The MODIFY CATALOG with REPLACE VOLUME command assumes that a subvolume contains a valid catalog if the subvolume contains the catalog table

TABLES (file code must be 581). The optional EXCLUDE catalog-list-2 clause specifies catalogs to be excluded from catalog-list-1.

```
REPLACE VOLUME volume-name [(node-name1 [,node-name2]
[EXCLUDE node-name3 [,node-name4] ] ) ]
```

specifies the volume name to be replaced in the catalog tables. SQL catalog tables contain file names, and those file names contain volume names. Volume names are changed based on the node-name list. The volume name specified in REPLACE VOLUME *volume-name* clause is replaced by the volume name specified in the WITH *volume-name* clause.

If the first *node-name* list is specified, SQL replaces the volume name only if it matches the volume name specified by *volume-name* and if the node name in the file name matches one of the node names specified in the *node-name* list. The optional EXCLUDE *node-name* list clause specifies node names to be excluded from the first *node-name* list.

volume-name must be preceded by a dollar '\$' character and must consist of at least one to six alphanumeric characters. The first character must be alphabetic. You can specify either uppercase or lowercase alphabetic characters; alphabetic characters are upshifted before the comparison and substitution process. The *volume-name* cannot include wild-card characters.

The first node-name list specifies a node name or a set of node names. *node-name* must be preceded by a backslash character. To specify a single node name, enter the name of the node. To specify multiple node names, use wild-card characters. You can use these wild-card characters:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates any name is acceptable. To specify all volumes, use either * or *
- ? Matches any single character

For example, \NOD* specifies all nodes that begin with \NOD. *NOD* matches NEWNODE, OLDNODE1, and NODE45. NOD? matches NOD1 and NODX, but not NOD or NOD48.

The maximum number of *node-names* that can be specified in this clause is 10 for each *node-name* list. For example, this clause is invalid because it has 11 node names in the first node-name list:

```
(\A*,\B*,\C*,\D*,\E*,\F*,\G*,\H*,\I*,\J*,\K* EXCLUDE \Z*)
```

If this clause is not specified, only the volume name is considered.

```
WITH volume-name
```

specifies the volume name to replace occurrences of the volume name specified in *replace-spec*.

volume-name must be preceded by a dollar '\$' character and consist of at least one to six alphanumeric characters. The first character must be alphabetic. Either uppercase or lowercase alphabetic characters can be specified; alphabetic characters are upshifted before the comparison and substitution process.

```
ALLOWERRORS [ OFF | ON | number-of-errors ]
```

determines handling of nonfatal errors. MODIFY reports two classes of errors: fatal errors and nonfatal errors. The MODIFY CATALOG command terminates immediately after reporting a fatal error. Nonfatal errors are handled depending on the value of the ALLOWERRORS option:

OFF	The MODIFY CATALOG with REPLACE VOLUME command terminates immediately after the first nonfatal error is encountered
-----	---

ON THE MODIFY CATALOG with REPLACE VOLUME command continues, regardless of the number of nonfatal errors encountered.

number-of-errors	The MODIFY CATALOG with REPLACE VOLUME command continues until the number of nonfatal errors exceeds value of <i>number-of-errors</i> .
------------------	---

If you specify ALLOWERRORS without ON, OFF, or the number of errors, ALLOWERRORS ON is the default. If you do not specify the ALLOWERRORS option, ALLOWERRORS OFF is the default.

When `MODIFY CATALOG` with `REPLACE VOLUME` continues processing after a nonfatal error has occurred, it advances to the next SQL catalog.

```
[ NO ] LISTALL
```

specifies how much detail MODIFY CATALOG with REPLACE VOLUME writes to the current OUT file. If LISTALL is specified, MODIFY CATALOG with REPLACE VOLUME reports the name of each SQL catalog considered for modification and whether or not the catalog was modified. If NO LISTALL is specified, only summary is reported.

LISTALL is the default option.

```

DETAIL [ MATCH | ALL ] REPORT [ TO EMS-Collector ]
                                     [ ON
                                     [ OFF

```

specifies that details about the MODIFY CATALOG with REPLACE VOLUME operation is to be sent in event messages to a valid EMS collector.

If MATCH is specified, details are reported about SQL catalog tables specified in target-spec that contain volume names that match the criteria specified in the

REPLACE clause. If ALL is specified, details are reported about all SQL catalog tables specified in target-spec irrespective of a match being identified with regards to volume name. MATCH is the default option.

For information on report options, see [REPORT Option](#) on page R-3.

CHECKONLY

specifies that the catalog tables specified in target-spec should be checked to see if they contain volume names that match the criteria specified in the REPLACE clause. No catalog tables are changed. The CHECKONLY option enables you to estimate the effect of running the MODIFY CATALOG with REPLACE VOLUME command before actually modifying the catalog tables.

Considerations-MODIFY CATALOG with REPLACE VOLUME

- You must be logged on as the super ID to run a MODIFY CATALOG with REPLACE VOLUME command unless you specify the CHECKONLY option. If you specify the CHECKONLY option, you must have authority to read the catalogs.
- SQL uses the TMF subsystem to protect the integrity of the database during the MODIFY operation. MODIFY CATALOG with REPLACE VOLUME command is not allowed inside a user-defined transaction.

One system-defined transaction is used for each set of catalog tables modified. If an error occurs while MODIFY CATALOG with REPLACE VOLUME is modifying a catalog table, the changes made to that catalog table and all other tables in the same catalog are backed out. However, changes that have been made to other catalogs during the same instance of the MODIFY CATALOG with REPLACE VOLUME command are not backed out. Thus, for one set of catalog tables, either the volume name will be changed in all the tables in the set or it will not be changed in any of the tables in the set.

- The MODIFY CATALOG with REPLACE VOLUME command is one of a set of commands that uses the MODIFY DICTIONARY utility. The other related commands are [MODIFY CATALOG with REPLACE NODENAME](#) (to replace node name in catalog), [MODIFY LABEL with REPLACE NODENUMBER](#) (to replace node number in file label), [MODIFY LABEL with REPLACE VOLUME](#) (to replace volume name in file label), and [MODIFY REGISTER Command](#) (to register user-defined catalogs in the local system catalog).
- If the volume name where the SQL catalog exists, has already been changed to a different name, then run the MODIFY LABEL with REPLACE VOLUME command on all catalog objects in the catalog sub-volume before executing the MODIFY CATALOG with REPLACE VOLUME command. This is required to update the volume name corresponding to the CATALOG field in the file label. The MODIFY CATALOG with REPLACE VOLUME cannot be run first because it will try to identify the CATALOG name from file label for record updation in catalog indexes.

However, if the volume name where the catalog exists has not been modified at the time of executing MODIFY DICTIONARY command, then consider executing MODIFY CATALOG with REPLACE VOLUME command first followed by MODIFY LABEL with REPLACE VOLUME.

For example, if a CATALOG exists on \$DATA00.CATVOL and the volume name has been changed to \$NEW00, the file labels of the catalog tables and indexes retain the CATALOG name field as \$DATA00. Running the MODIFY CATALOG with REPLACE VOLUME command on these catalog tables and indexes fails because it tries to identify the catalog \$DATA00.CATVOL for updating certain records in indexes on catalog tables such as IXINDE01. Therefore, the update fails because the \$DATA00 name field has changed to \$NEW00. Hence consider running the MODIFY LABEL with REPLACE VOLUME command on these catalog objects such that the file labels are updated with the new catalog name. This can be followed by the MODIFY CATALOG with REPLACE VOLUME command.

- The MODIFY CATALOG with REPLACE VOLUME command locks one catalog table at a time. The file labels of the catalog tables are not locked. Do not request DDL or update operations until the volume names are modified, including operations on partitions and on dependent objects on remote nodes.
- The MODIFY DICTIONARY utility does not handle the following:
 - Remote nodes. If you specify a remote file name or catalog name, the MODIFY command reports a nonfatal error. Therefore, each node with dependent objects or partitioned objects must have a version of NonStop SQL/MP that supports MODIFY CATALOG commands.
 - User-defined SQL object files. For example, the MODIFY CATALOG with REPLACE VOLUME command does not modify a volume name stored in a column of a user-defined table.
 - Names stored in SQL object program files. SQL object programs can refer to a SQL object by using either a DEFINE or the Guardian name of the object. If DEFINES are used, both the DEFINE name and the associated Guardian name of the SQL object are stored in the object program file-not in the file label. If Guardian names are used, the Guardian names are stored in the object program file in internal network form.
 - Dependent objects. The MODIFY CATALOG with REPLACE VOLUME command does not modify information about dependent objects that reside in other catalogs unless the other catalogs are specified. You must be aware of how the database is distributed. Document the MODIFY commands that need to be run, the nodes they need to be run on, before they are used. Prepare scripts that run the necessary MODIFY commands. When you add a new dependent object to the database, update the scripts. While the node is in a consistent state, you can use the DISPLAY USE OF command to locate dependent objects. After MODIFY commands are run, use the VERIFY utility to verify that the database is in a consistent state.

- Partitioned objects. For a partitioned SQL object, each volume that contains a partition of the object must be specified separately. MODIFY CATALOG does not automatically modify information about all partitions of a partitioned object. You must be aware of how the database is partitioned. While the node is in a consistent state, SELECT from the PARTNS partitions table to locate other partitions. Prepare scripts that run the necessary MODIFY commands. After running the MODIFY commands, use the VERIFY utility to verify that the database is in a consistent state.
- The MODIFY CATALOG with REPLACE VOLUME command does not mark SQL object programs as invalid in either the catalog or the object program file label.
- The MODIFY CATALOG with REPLACE VOLUME command does not change the redefinition timestamp in either the catalog or the file label.
- You can use DEFINE names in programs to specify names of catalogs, tables, views, indexes, partitions, and other programs. The current DEFINE set at the time the program is SQL compiled is saved in the object program file. If objects or object programs specified by the DEFINES are moved between the time that the program is SQL compiled and the time that the program is run, the DEFINES must be changed to reflect the new location of the objects and object programs. This guideline is true regardless of how the database was moved and regardless of whether the MODIFY DICTIONARY utility was used to modify the node names and numbers.

The MODIFY DICTIONARY utility does not modify volume names in the DEFINE set stored in the object program file. After you have changed the DEFINES, if automatic recompilation is enabled, the programs are automatically recompiled using the new DEFINES.

- The MODIFY DICTIONARY utility is not intended to correct the situation where a disk containing objects is moved to a new node, but the disk containing the associated catalog is not moved to the new node.
- The MODIFY CATALOG with REPLACE VOLUME command returns a nonfatal versioning error if an SQL catalog has a version newer than the version of MODIFY that is accessing it, or if the catalog has a version newer than the version of SQL/MP system software on the node where the catalog resides.

Examples-MODIFY CATALOG with REPLACE VOLUME

These examples illustrate the use of the REPLACE clause.

- In this example, the volume name \$DATA is replaced with the volume name \$NEW if the first three characters of the node name are \DA:

```
REPLACE VOLUME $DATA (\DA*) WITH $NEW
```

- In this REPLACE clause, the volume name \$DATA is replaced with the volume name \$NEW if the node name is either \SAM or \IND or if the first three characters of the node name are \DA:

```
REPLACE VOLUME $DATA (\SAM,\IND,\DA*) WITH $NEW
```

- In this REPLACE clause, the volume name \$DATA will be replaced with the volume name \$NEW if the node name is anything other than \IND:

```
REPLACE VOLUME $DATA (\* EXCLUDE \IND) WITH $NEW
```

- When you use the CHECKONLY option, note that the amount of information written to the current OUT file depends on whether LISTALL or NO LISTALL is specified. The MODIFY CATALOG command produces this display for each catalog when requested with the LISTALL option:

```
Checking catalog \SYS.$VOL.CAT1.
```

```
--- \SYS.$VOL.CAT1 was modified.
```

```
Checking catalog \SYS.$VOL.CAT2.
```

```
--- \SYS.$VOL.CAT2 was not modified.
```

- The following summary information is included whether you specify LISTALL or NO LISTALL:

```
Summary Information:
```

```
nnn catalog(s) require modification.
```

```
nnn catalog(s) do not require modification.
```

MODIFY LABEL

The MODIFY LABEL command has two options that can be specified as the *replace-spec*:

- [MODIFY LABEL with REPLACE NODENUMBER](#)
- [MODIFY LABEL with REPLACE VOLUME](#)

MODIFY LABEL with REPLACE NODENUMBER

The MODIFY LABEL with REPLACE NODENUMBER command modifies node numbers stored in file labels of SQL objects and SQL object programs on the local node. The file label of an SQL object or object program contains names in internal network form and therefore always contains one or more node numbers.

This command is intended for use when physically moving a disk from one node to another node, or when system loading a node with a new node name or number. These changes are not automatically reflected in the catalogs and file labels. When this happens, the internal consistency of the database is lost; catalogs (and objects they describe) cannot be accessed. The MODIFY commands (LABEL, CATALOG, and REGISTER) enables you to change the node number or node name to reflect the new information.

SQL objects include SQL catalog tables, user-defined tables, table partitions, indexes, index partitions, views, and collations.

```

MODIFY [DICTIONARY] LABEL target-spec replace-spec

    WITH node-spec [ [,] option ] ... ;

target-spec is:

    simple-fileset-list-1 [EXCLUDE simple-fileset-list-2]

replace-spec is:

    REPLACE NODENUMBER node-spec[(volumeset[,volumeset]
        [EXCLUDE volumeset [,volumeset] ] ) ]

option is:

    [ | ALLOWERRORS [ OFF | ON | number-of-errors ] | ]
    [ | [ NO ] LISTALL | ]
    [ | DETAIL [ MATCH | ALL ] REPORT [ TO EMS-Collector ] | ]
    [ | [ ON ] | ]
    [ | [ OFF ] | ]
    [ | CHECKONLY | ]

simple-fileset-list-n (where n = 1,2) is:

    { fileset
      { ( fileset [ , fileset ] ... ) }
    }

node-spec is:

    { ( node-name , [ node-number ] )
      { ( node-number , [ node-name ] ) }
    }

```

simple-fileset-list-1 [EXCLUDE *simple-fileset-list-2*]

identifies one or more SQL objects and object programs whose file labels will be considered for modification. File labels are modified only if they contain node numbers that match the criteria specified in the REPLACE clause. Files included in *simple-fileset-list-1* that are not SQL objects or SQL program files are ignored.

The optional EXCLUDE *simple-fileset-list-2* clause specifies files to be excluded from *simple-fileset-list-1*.

fileset is a Guardian name in which wild-card characters can be used to specify volumes, subvolumes, files, and objects. You cannot use wild-card characters in node names.

The wild-card characters you can use are:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates that any name is acceptable. To specify all files on all volumes, use either \$*.* or *.*.
- ? Matches any single character

For example, *VOL* matches NEWVOL, OLDVOL1, and VOL45. \$VOL1.SUBV1.* specifies all files on subvolume SUBV1 of volume \$VOL1, and \SYS1.*.SUBV1.* specifies all files on all subvolumes named SUBV1 on any volume of node \SYS1. TABLE? matches TABLE1 and TABLEX, but not TABLE or TABLE48.

You can also specify a logical DEFINE name as a fileset. File labels for dependent objects and partitions are not considered for modification unless the dependent object or partition is specified in *simple-fileset-list-1*. For example, if \$A.B.T1 is a table that has a dependent index, specifying \$A.B.T1 results in only the file label of the table being considered; the file label of the index is not considered.

```
REPLACE NODENUMBER node-spec [ ( volumeset [,volumeset]
[EXCLUDE volumeset [,volumeset] ] ) ]
```

from one to seven alphanumeric characters. The first specifies the node number to replace in the file labels. The node number is replaced by the node number specified in the WITH *node-spec* clause.

If the first *volumeset* list is specified, SQL replaces the node number only if the node number in the file label matches the node number specified by *node1* and the volume name of the file label matches one of the volume names specified in the first *volumeset* list. The optional EXCLUDE *volumeset* list clause specifies volumes to be excluded from the first *volumeset* list.

node-spec specifies a *node-number* and *node-name* pair. *node-name* specifies a node name, which must be preceded by a character must be alphabetic. Either uppercase or lowercase alphabetic characters can be specified. *node-number* specifies a node number in the range from 0 to 254.

The value of *node-spec* need not match the node number assigned to the local node executing the MODIFY command.

volumeset indicates a volume or a set of volumes. To specify a single volume, enter the name of the volume. To specify multiple volumes use wild-card characters. You can use these wild-card characters:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates any name is acceptable. To specify all volumes, use either \$* or *
- ? Matches any single character

For example, \$DAT* specifies all volumes that begin with \$DAT. *VOL* matches NEWVOL, OLDVOL1, and VOL45. VOL? matches VOL1 and VOLX but not VOL or VOL48.

The maximum number of *volumesets* that can be specified in this clause is 10 for each *volume* list. For example, this clause is invalid because it has 11 *volumesets* in the first *volume* list:

```
( $A* , $B* , $C* , $D* , $E* , $F* , $G* , $H* , $I* , $J* , $K* EXCLUDE $AA* )
```

If this clause is not specified, only the node number is considered.

WITH *node-spec*

specifies the node number to replace occurrences of the node number specified by *replace-spec*.

node-spec specifies a *node-number*, *node-name*, or both. *node-number* specifies a node number in the range from 0 to 254. *node-name* specifies a node name, which must be preceded by a backslash character and can consist of from one to seven alphanumeric characters. The first character must be alphabetic. Either uppercase or lowercase alphabetic character can be specified.

The MODIFY LABEL with REPLACE NODENUMBER command must be able to identify a valid node number from the information given. The MODIFY LABEL with REPLACE NODENUMBER command determines the node number:

- If you specify a recognized node name and do not specify a node number, the corresponding node number is used.
- If you specify only a node name and the name is not known, SQL returns a fatal error and the command terminates. An unknown name cannot be mapped to a known node number.
- If the specified node name and node number are both unknown, SQL returns a warning message noting that the node name and number are unknown and that the node name is being ignored. It then proceeds with the request, using the specified node number.
- If both a node name and number are specified, and both refer to a known node, both must refer to the same node. Otherwise, SQL returns an error.

The value of *node-spec* need not match the node number assigned to the local node executing the MODIFY command.

ALLOWERRORS [OFF | ON | *number-of-errors*]

determines handling of nonfatal errors. MODIFY LABEL with REPLACE NODENUMBER reports two classes of errors: fatal errors and nonfatal errors. The MODIFY LABEL with REPLACE NODENUMBER command terminates immediately after reporting a fatal error. Nonfatal errors are handled depending on the value of the ALLOWERRORS option:

OFF	The MODIFY LABEL with REPLACE NODENUMBER command terminates immediately after the first nonfatal error is encountered.
ON	THE MODIFY LABEL with REPLACE NODENUMBER command continues, regardless of the number of nonfatal errors encountered.
<i>number-of-errors</i>	The MODIFY LABEL with REPLACE NODENUMBER command continues until the number of nonfatal errors exceeds the value of <i>number-of-errors</i> .

If you specify ALLOWERRORS without ON, OFF, or the number of errors, ALLOWERRORS ON is the default.

If you do not specify the ALLOWERRORS option, ALLOWERRORS OFF is the default.

The description of each MODIFY LABEL with REPLACE NODENUMBER error states whether the error is fatal or nonfatal.

When MODIFY LABEL with REPLACE NODENUMBER continues processing after a nonfatal error has occurred, it advances to the next SQL object or object program.

[NO] LISTALL

specifies how much information MODIFY LABEL with REPLACE NODENUMBER writes to the current OUT file. If LISTALL is specified, MODIFY reports the name of each SQL object and object program whose label was considered for modification and indicates whether or not the object or program was modified. If NO LISTALL is specified, only summary information is reported.

LISTALL is the default.

DETAIL [MATCH | ALL] REPORT [TO *EMS-Collector*
[ON]
[OFF]

specifies that detailed about the MODIFY LABEL with REPLACE NODENUMBER operation is to be sent in event messages to a valid EMS collector.

If MATCH is specified, detail is reported about SQL objects and object programs specified in *target-spec* that contain node numbers matching the criteria

specified in the REPLACE clause. If ALL is specified, detail is reported about all SQL objects and object programs specified in *target-spec*.

MATCH is the default.

For information on report options, see [REPORT Option](#) on page R-3.

CHECKONLY

specifies that file labels specified by *target-spec* should be checked to see if they contain node numbers that match the criteria specified in the REPLACE clause. No file labels are modified. The CHECKONLY option enables you to estimate the effect of running the MODIFY LABEL with REPLACE NODENUMBER command before actually modifying the file labels.

The amount of information written to the current OUT file depends on whether LISTALL or NO LISTALL is specified.

Considerations—MODIFY LABEL with REPLACE NODENUMBER

- You must be logged on as the super ID to run a MODIFY LABEL with REPLACE NODENUMBER command unless you specify the CHECKONLY option. If the MODIFY LABEL with REPLACE NODENUMBER CHECKONLY option is specified, the user must have authority to read the SQL objects and object programs.
- NonStop SQL/MP uses the TMF subsystem to protect the integrity of the database during the MODIFY operation. MODIFY LABEL with REPLACE NODENUMBER commands are not allowed inside a user-defined transaction.

One system-defined transaction is used for each SQL object file label modified. If an error occurs while MODIFY LABEL with REPLACE NODENUMBER is in the middle of modifying a file label of an SQL object, the changes made to that particular label are backed out. Changes made to other file labels by the same instance of the MODIFY LABEL with REPLACE NODENUMBER command are not backed out. Thus, if an error occurs, some labels might have been changed and others might not have been.

SQL object program file labels are not modified within a TMF transaction. This means that if an error occurs while modifying the file label of an object program file, the label could be left in an inconsistent state.

- The MODIFY LABEL with REPLACE NODENUMBER command is one of a set of commands that uses the MODIFY DICTIONARY utility. The other related commands are [MODIFY LABEL with REPLACE VOLUME](#) (to replace volume name in file label), [MODIFY CATALOG with REPLACE VOLUME](#) (to replace volume name in catalog), [MODIFY CATALOG with REPLACE NODENAME](#) (to replace node name in catalog), and [MODIFY REGISTER Command](#) (to register user-defined catalogs in the local system catalog).
- To minimize unnecessary searching, make your *target-spec* clause as specific as possible. For example, if SQL objects reside only on subvolumes whose names

begin with SQL, specifying \$VOL1.SQL *.* is more efficient than specifying \$VOL1.*.*. The MODIFY LABEL with REPLACE NODENUMBER command would not have to search for SQL objects on other subvolumes.

- Multiple MODIFY commands (including LABEL, CATALOG, and REGISTER commands) can be run concurrently on the same node as long as each command is processing a different set of files or catalogs. For example, if the node number is changed on a node that has an SQL database spread out over five volumes, five MODIFY LABEL with REPLACE NODENUMBER commands can be started concurrently, each specifying a different volume to be modified. Note that in such a case, because the node name did not change, it would not be necessary to run any MODIFY CATALOG commands.
- MODIFY LABEL with REPLACE NODENUMBER locks one file label at a time. The file itself is not locked. There is nothing to prevent the user from accessing a partially modified data dictionary. The user should refrain from using the database, including partitions and dependent objects on remote nodes, until the node numbers have been modified.
- The MODIFY DICTIONARY utility does not handle:
 - Remote nodes. If you specify a remote file name, the MODIFY LABEL with REPLACE NODENUMBER command reports a nonfatal error. Each node with dependent objects or partitioned objects must have a version of NonStop SQL/MP that supports MODIFY LABEL with REPLACE NODENUMBER commands.
 - User-defined SQL object files. For example, MODIFY LABEL with REPLACE NODENUMBER does not modify a node number stored in a column of a user-defined table.
 - Node numbers stored in SQL object program files.
 - Node numbers in Enscribe file labels. In Enscribe file labels, local file names are stored as local names and thus do not contain a node number. However, a reference to an alternate key file or a partition file does include a node number if the alternate key file or partition file is stored on a different node.

If a disk containing an alternate key file or a partition file is moved to a different node, the FUP ALTER command can be used to change the Enscribe file label that references the file that moved. Note that changes must be made to the Enscribe file labels that point to the disk that moved, not to the alternate key file label or partition file label that resides on the disk that moved.

- Dependent objects. Node numbers in the file labels of dependent objects are not modified unless the dependent object is specified in the MODIFY LABEL with REPLACE NODENUMBER command.

For example, suppose that a table T1 resides on the \SYS1.\$DB1.OBJECTS subvolume and is registered in the \SYS1.\$DB1.CAT catalog, and its dependent index I1 resides on the \SYS2.\$DBS.OBJECTS subvolume and is registered in the \SYS2.\$DBS.CAT catalog. Suppose that the \$DBS disk is

moved from \SYS2 to \SYS1. The database is left in an inconsistent state if only these commands are run:

```
>> MODIFY LABEL $DBS.*.*
+> REPLACE NODENUMBER \SYS2 ($DBS) WITH \SYS1;
>> MODIFY CATALOG $DBS.CAT
+> REPLACE NODENAME \SYS2 ($DBS) WITH \SYS1;
```

One example of an inconsistency that exists is that the \SYS1.\$DB1.CAT.USAGES table will indicate that the USINGOBJNAME of the index is \SYS2.\$DBS.OBJECTS.I1, although \$DBS is now on \SYS1.

- Partitioned objects. Node numbers in the file labels of partitions of tables and indexes are not modified unless those partitions are specified in the MODIFY LABEL with REPLACE NODENUMBER command.

For a partitioned SQL object, each volume that contains a partition of the object must be specified separately. MODIFY LABEL with REPLACE NODENUMBER does not automatically modify information about all partitions of a partitioned object.

It is the responsibility of the user to know how the database is distributed and partitioned. Document the MODIFY commands that need to be run—and what nodes they need to be run on—before they are needed. Prepare scripts that run the necessary MODIFY commands. When you add a new dependent object to the database, update the scripts.

While the system is in a consistent state, you can use the DISPLAY USE OF command to locate dependent objects. After MODIFY commands have been run, you can use the VERIFY utility to verify that the database is in a consistent state.

While the system is in a consistent state, issue a SELECT from the PARTNS partitions table to locate partitions. Prepare scripts that run the necessary MODIFY commands. After MODIFY commands are run, use the VERIFY utility to verify that the database is in a consistent state.

- MODIFY LABEL with REPLACE NODENUMBER does not mark SQL object programs as invalid in either the catalog or in the object program file label.
- MODIFY LABEL with REPLACE NODENUMBER does not change the redefinition timestamp in either the catalog or the file label.
- DEFINE names can be used in programs to specify the names of catalogs, tables, views, indexes, partitions, and other programs. The current DEFINE set at the time the program is SQL compiled is saved in the object program file. If objects or object programs specified by the DEFINES are moved between the time that the program is SQL compiled and the time that the program is run, the DEFINES must be changed to reflect the new location of the objects and object programs. This instruction is true regardless of how the database was moved or whether the MODIFY DICTIONARY utility was used to modify the node names and numbers.

The MODIFY DICTIONARY utility does not modify the node names in the DEFINE set stored in the object program file.

- After the DEFINES are changed by the user, if automatic recompilation is enabled the programs are automatically recompiled using the new DEFINES.
- A catalog and the objects registered in it must be on the same node. However, it is possible for an object to be on a different disk than its catalog. The MODIFY DICTIONARY utility is not intended to correct the situation where a disk containing objects is moved to a new node, but the disk containing the associated catalog is not moved to the new node.
- The MODIFY LABEL with REPLACE NODENUMBER command returns a nonfatal versioning error if an SQL object is newer than the version of MODIFY DICTIONARY accessing it, if the PCV of an SQL program is newer than the version of MODIFY that is accessing it, or if the object or program has a version newer than the version of SQL/MP system software on the node where the object or program resides.

Examples—MODIFY LABEL with REPLACE NODENUMBER

The first group of examples show the REPLACE clause.

- In this REPLACE clause, node number 24 is replaced with node number 75 if the first three characters of the volume name are \$DA:

```
REPLACE NODENUMBER 24 ($DA*) WITH 75
```

- In this REPLACE clause, node number 100 is replaced with node number 175 if the volume name is either \$SAM or \$CAT or if the first three characters of the volume name are \$DA:

```
REPLACE NODENUMBER 100 ($SAM,$CAT,$DA*) WITH 175
```

- In this REPLACE clause, node number 100 is replaced with node number 175 if the volume name is anything other than \$SYSTEM:

```
REPLACE NODENUMBER 100 ($* EXCLUDE $SYSTEM) WITH 175
```

- In this REPLACE clause, node number 24 is replaced with node number 100:

```
REPLACE NODENUMBER 24 WITH 100
```

- In this REPLACE clause, node number 24 is replaced with the node number associated with the node name \SQL:

```
REPLACE NODENUMBER 24 WITH \SQL
```

- In this REPLACE clause, node number 24 is replaced with the node number 100 if 100 is the node number associated with the node name \SQL:

```
REPLACE NODENUMBER 24 WITH (\SQL,100)
```

- In this REPLACE clause, the node number associated with the node name \SQLNLS is replaced with the node number associated with the node name \SQL:

```
REPLACE NODENUMBER \SQLNLS WITH \SQL
```

- In this REPLACE clause, node number 50 is replaced with the node number 100 if 50 is the node number associated with the node name \SQLNLS and 100 is the node number associated with the node name \SQL:

```
REPLACE NODENUMBER (\SQLNLS,50) WITH (\SQL,100)
```

- This example lists the display for each object when CHECKONLY is requested with the LISTALL option:

```
Checking \SYS.$VOL.SUBVOL.T1 label.
--- \SYS.$VOL.SUBVOL.T1 label requires modification.
Checking \SYS.$VOL.SUBVOL.T2 label.
--- \SYS.$VOL.SUBVOL.T2 label does not require modification.
```

- This example on summary information is displayed for CHECKONLY if you specify either LISTALL or NO LISTALL:

```
Summary Information:
    nnn label(s) require modification.
    nnn label(s) do not require modification.
```

- This MODIFY LABEL with REPLACE NODENUMBER command produces this display when requested with the LISTALL option:

```
Checking \SYS.$VOL.SUBVOL.T1 label.
--- \SYS.$VOL.SUBVOL.T1 label was modified.
Checking \SYS.$VOL.SUBVOL.T2 label.
--- \SYS.$VOL.SUBVOL.T2 label was not modified.
```

- This example on summary information is included whether you specify LISTALL or NO LISTALL:

```
Summary Information:
    nnn label(s) modified.
    nnn label(s) not modified.
```

MODIFY LABEL with REPLACE NODENUMBER and Partitioned Objects

This example shows the usage of MODIFY LABEL with REPLACE NODENUMBER when you move partitioned objects.

Suppose that you have table T1 with one partition at \A.\$DA1.SQL.T1, a second partition at \A.\$DB1.SQL.T1, and a third partition at \C.\$DC1.SQL.T1. The catalogs where the partitions are registered contain references to the other partitions. For example, the catalog where \A.\$DA1.SQL.T1 is registered contains a reference in the PARTNS catalog table to all three partitions, and likewise for the catalogs where \A.\$DB1.SQL.T1 and \C.\$DC1.SQL.T1 are registered.

This SELECT statements illustrate this point:

```
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \A.$DA1.CATSUBV.PARTNS;
FILENAME          PARTITIONNAME          CATALOGNAME
-----
\A.$DA1.SQL.T1    \A.$DA1.SQL.T1    \A.$DA1.CATSUBV
\A.$DA1.SQL.T1    \A.$DB1.SQL.T1    \A.$DB1.CATSUBV
\A.$DA1.SQL.T1    \C.$DC1.SQL.T1    \C.$DC1.CATSUBV
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \A.$DB1.CATSUBV.PARTNS;
FILENAME          PARTITIONNAME          CATALOGNAME
-----
\A.$DB1.SQL.T1    \A.$DA1.SQL.T1    \A.$DA1.CATSUBV
\A.$DB1.SQL.T1    \A.$DB1.SQL.T1    \A.$DB1.CATSUBV
\A.$DB1.SQL.T1    \C.$DC1.SQL.T1    \C.$DC1.CATSUBV
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \C.$DC1.CATSUBV.PARTNS;

FILENAME          PARTITIONNAME          CATALOGNAME
-----
\C.$DC1.SQL.T1    \A.$DA1.SQL.T1    \A.$DA1.CATSUBV
\C.$DC1.SQL.T1    \A.$DB1.SQL.T1    \A.$DB1.CATSUBV
\C.$DC1.SQL.T1    \C.$DC1.SQL.T1    \C.$DC1.CATSUBV
```

Now suppose that the \$DB1 volume is moved from the \A node (node number 101) to the \B node (node number 102). The MODIFY DICTIONARY commands needed to make the NonStop SQL/MP database consistent after the move are shown:

On node \B, to modify the volume moved from \A to \B:

```
>> MODIFY LABEL $DB1.*.*
+> REPLACE NODENUMBER 101 ($DB1) WITH 102;
>> MODIFY CATALOG $DB1.CATSUBV

+> REPLACE NODENAME \A ($DB1) WITH \B;
>> MODIFY REGISTER CATALOG $DB1.CATSUBV;
```

On node \A, to modify the references to the partition moved from \A to \B:

```
>> MODIFY LABEL $DA1.*.*
+> REPLACE NODENUMBER 101 ($DB1) WITH 102;
>> MODIFY CATALOG $DA1.CATSUBV
+> REPLACE NODENAME \A ($DB1) WITH \B;
```

On node \C, to modify the references to the partition moved from \A to \B:

```
>> MODIFY LABEL $DC1.*.*
+> REPLACE NODENUMBER 101 ($DB1) WITH 102;
>> MODIFY CATALOG $DC1.CATSUBV
+> REPLACE NODENAME \A ($DB1) WITH \B;
```

Notice that the REPLACE clauses are specified as:

```
REPLACE NODENUMBER 101 ($DB1)
REPLACE NODENAME \A ($DB1).
```

The (\$DB1) part is necessary to change all references from \A.\$DB1.SQL.T1 to \B.\$DB1.SQL.T1 while leaving intact all references to the \A.\$DA1.SQL.T1 partition that remains on \A.

After the MODIFY commands are run, the information in the PARTNS catalog tables looks like this:

```
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \A.$DA1.CATSUBV.PARTNS;
```

FILENAME	PARTITIONNAME	CATALOGNAME
\A.\$DA1.SQL.T1	\A.\$DA1.SQL.T1	\A.\$DA1.CATSUBV
\A.\$DA1.SQL.T1	\B.\$DB1.SQL.T1	\B.\$DB1.CATSUBV
\A.\$DA1.SQL.T1	\C.\$DC1.SQL.T1	\C.\$DC1.CATSUBV

```
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \B.$DB1.CATSUBV.PARTNS;
```

FILENAME	PARTITIONNAME	CATALOGNAME
\B.\$DB1.SQL.T1	\A.\$DA1.SQL.T1	\A.\$DA1.CATSUBV
\B.\$DB1.SQL.T1	\B.\$DB1.SQL.T1	\B.\$DB1.CATSUBV
\B.\$DB1.SQL.T1	\C.\$DC1.SQL.T1	\C.\$DC1.CATSUBV

```
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \C.$DC1.CATSUBV.PARTNS;
```

FILENAME	PARTITIONNAME	CATALOGNAME
\C.\$DC1.SQL.T1	\A.\$DA1.SQL.T1	\A.\$DA1.CATSUBV
\C.\$DC1.SQL.T1	\B.\$DB1.SQL.T1	\B.\$DB1.CATSUBV
\C.\$DC1.SQL.T1	\C.\$DC1.SQL.T1	\C.\$DC1.CATSUBV

MODIFY LABEL with REPLACE VOLUME

Note. The MODIFY LABEL with REPLACE VOLUME command is supported on systems running J06.04 and later J-series RVUs and H06.15 and later H-series RVUs only.

The MODIFY LABEL with REPLACE VOLUME command modifies volume names stored in file labels of SQL objects and SQL object programs on the local node. The file label of an SQL object or object program contains names in internal network form and therefore always contains one or more volume names. This command is used when physically moving a disk from one node in your network to another node which has a similar disk volume name, or when system loading a disk with a new disk volume name. When this happens, changes are not automatically reflected in the catalogs and file labels. The internal consistency of the database is lost; catalogs (and objects they describe) cannot be accessed. The MODIFY commands (LABEL, CATALOG) enable you to change the database to reflect the new information.

SQL objects include SQL catalog tables, user-defined tables, table partitions, indexes, index partitions, views, and collations.

```

MODIFY [DICTIONARY] LABEL target-spec replace-spec
WITH volume-name [ [,] option ] ... ;

target-spec is:

simple-fileset-list-1 [ EXCLUDE simple-fileset-list-2 ]

replace-spec is:

REPLACE VOLUME volume-name [ ( node-name1 [,node-name2]
[EXCLUDE node-name3 [,node-name4]... ] ) ]

option is:
[ | ALLOWERRORS [ OFF | ON | number-of-errors ] | ]
[ | [ NO ] LISTALL | ]
[ | DETAIL [ MATCH | ALL ] REPORT [ TO EMS-Collector ] | ]
[ | [ ON ] | ]
[ | [ OFF ] | ]
[ | CHECKONLY | ]

simple-fileset-list-n (where n = 1,2) is:
{ fileset }
{ ( fileset [ , fileset ] ... ) }

```

simple-fileset-list-1 [EXCLUDE *simple-fileset-list-2*]

identifies one or more SQL objects and object programs whose file labels will be considered for modification. File labels are modified only if they contain volume names that match the criteria specified in the REPLACE clause. Files included in *simple-fileset-list-1* that are not SQL objects or SQL program files are ignored.

The optional EXCLUDE *simple-fileset-list-2* clause specifies files to be excluded from *simple-fileset-list-1*.

fileset is a Guardian name in which wild-card characters can be used to specify volumes, subvolumes, files, and objects. You cannot use wild-card characters in node names. The wild-card characters you can use are:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates that any name is acceptable. To specify all files, use either \$*.* or *.*.
- ? Matches any single character

For example, *VOL* matches NEWVOL, OLDVOL1, and VOL45. \$VOL1.SUBV1.* specifies all files on subvolume SUBV1 of volume \$VOL1, and \SYS1.*.SUBV1.* specifies all files on all subvolumes named SUBV1 on any volume of node \SYS1. TABLE? matches TABLE1 and TABLEX, but not TABLE or TABLE48.

You can also specify a logical DEFINE name as a fileset. File labels for dependent objects and partitions are not considered for modification unless the dependent object or partition is specified in simple-fileset-list-1. For example, if \$A.B.T1 is a table that has a dependent index, specifying \$A.B.T1 results in only the file label of the table being considered; the file label of the index is not considered.

```
REPLACE VOLUME volume-name [(node-name1 [,node-name2]
[EXCLUDE node-name3 [,node-name4] ] ) ]
```

specifies the volume name to be replaced in the file label. SQL file labels contain file names, and those file names contain volume names. Volume names are changed based on the node-name list. The volume name specified in REPLACE VOLUME *volume-name* clause is replaced by the volume name specified in the WITH *volume-name* clause.

If the first node-name list is specified, SQL replaces the volume name only if it matches the volume name specified by *volume-name* and if the node name in the file name matches one of the node names specified in the node-name list. The optional EXCLUDE node-name list clause specifies node names to be excluded from the first node-name list.

volume-name must be preceded by a dollar '\$' character and must consist of at least one to six alphanumeric characters. The first character must be alphabetic. You can specify either uppercase or lowercase alphabetic characters; alphabetic characters are upshifted before the comparison and substitution process. The volume-name cannot include wild-card characters.

The first node-name list specifies a node name or a set of node names.

node-name must be preceded by a backslash character. To specify a single node name, enter the name of the node. To specify multiple node names, use wild-card characters. You can use these wild-card characters:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates any name is acceptable. To specify all files, use either * or *
- ? Matches any single character

For example, \NOD* specifies all nodes that begin with \NOD. *NOD* matches NEWNODE, OLDNODE1, and NODE45. NOD? matches NOD1 and NODX, but not NOD or NOD48.

The maximum number of node-names that can be specified in this clause is 10 for each node-name list. For example, this clause is invalid because it has 11 node names in the first node-name list:

```
(\A*,\B*,\C*,\D*,\E*,\F*,\G*,\H*,\I*,\J*,\K* EXCLUDE \Z*)
```

If this clause is not specified, only the volume name is considered.

WITH *volume-name*

specifies the volume name to replace occurrences of the volume name specified in *replace-spec*.

volume-name must be preceded by a dollar '\$' character and consist of at least one to six alphanumeric characters. The first character must be alphabetic. Either uppercase or lowercase alphabetic characters can be specified; alphabetic characters are upshifted before the comparison and substitution process.

ALLOWERRORS [OFF | ON | *number-of-errors*]

determines handling of nonfatal errors. MODIFY reports two classes of errors: fatal errors and nonfatal errors. The MODIFY LABEL with REPLACE VOLUME command terminates immediately after reporting a fatal error. Nonfatal errors are handled depending on the value of the ALLOWERRORS option:

OFF	The MODIFY LABEL with REPLACE VOLUME command terminates immediately after the first nonfatal error is encountered.
ON	THE MODIFY LABEL with REPLACE VOLUME command continues, regardless of the number of nonfatal errors encountered.
<i>number-of-errors</i>	The MODIFY LABEL with REPLACE VOLUME command continues until the number of nonfatal errors exceeds value of <i>number-of-errors</i> .

If you specify ALLOWERRORS option without ON, OFF, or the number of errors, ALLOWERRORS ON is the default. If you do not specify the ALLOWERRORS option, ALLOWERRORS OFF is the default.

When the MODIFY LABEL with REPLACE VOLUME command continues processing after a nonfatal error has occurred, it advances to the next SQL file.

[NO] LISTALL

specifies the detail MODIFY LABEL with REPLACE VOLUME writes to the current OUT file. If LISTALL is specified, MODIFY LABEL with REPLACE VOLUME reports the name of each SQL file label considered for modification and whether or not the file label was modified. If NO LISTALL is specified, only the summary is reported.

LISTALL is the default option.

```
DETAIL [ MATCH | ALL ] REPORT [ TO EMS-Collector ]
                                     [ ON          ]
                                     [ OFF         ]
```

specifies the details about the MODIFY LABEL with REPLACE VOLUME operation that is to be sent in event messages to a valid EMS collector.

If MATCH is specified, details are reported about SQL file labels specified in target-spec that contain volume names that match the criteria specified in the REPLACE clause. If ALL is specified, details are reported about all SQL file labels specified in target-spec irrespective of a match being identified with regards to volume name. MATCH is the default option.

CHECKONLY

specifies that the file labels specified in target-spec should be checked to see if they contain volume names that match the criteria specified in the REPLACE clause. No file labels are changed. The CHECKONLY option enables you to estimate the effect of running the MODIFY LABEL with REPLACE VOLUME command before modifying the SQL file labels.

Considerations-MODIFY LABEL with REPLACE VOLUME

- You must be logged on as the super ID to run a MODIFY LABEL with REPLACE VOLUME command unless you specify the CHECKONLY option. If you specify the CHECKONLY option, you must have authority to read the catalogs.
- SQL uses the TMF subsystem to protect the integrity of the database during the MODIFY operation. The MODIFY LABEL with REPLACE VOLUME command is not allowed inside a user-defined transaction.

One system-defined transaction is used for each SQL object file label modified. If an error occurs while MODIFY LABEL with REPLACE VOLUME is modifying a file label, the changes made to that particular file label is backed out. Changes made to other file labels by the same instance of MODIFY LABEL with REPLACE VOLUME command are not backed out. Thus, if an error occurs, some labels might have been changed and others might not have been changed.

- The MODIFY LABEL with REPLACE VOLUME command is one of a set of commands that uses the MODIFY DICTIONARY utility. The other related commands are [MODIFY CATALOG with REPLACE NODENAME](#) (to replace node name in catalog), [MODIFY LABEL with REPLACE NODENUMBER](#) (to replace node number in file label), [MODIFY CATALOG with REPLACE VOLUME](#) (to replace volume name in catalog), and [MODIFY REGISTER Command](#) (to register user-defined catalogs in the local system catalog).
- If the volume name where the SQL catalog exists has already been changed to a different name, then run the MODIFY LABEL with REPLACE VOLUME command on all catalog objects in the catalog sub-volume before running the MODIFY CATALOG with REPLACE VOLUME command. This is required to update the volume name corresponding to the CATALOG field in the file label. The MODIFY

CATALOG with REPLACE VOLUME command cannot be run first because it will try to identify the CATALOG name from the file label to update the records in catalog indexes.

However, if the volume name where the catalog exists has not been modified at the time of running the MODIFY DICTIONARY command, consider executing MODIFY CATALOG with REPLACE VOLUME command first followed by MODIFY LABEL with REPLACE VOLUME command.

For example, if a catalog exists on \$DATA00.CATVOL and the volume name has been changed to \$NEW00, the file labels of the catalog tables and indexes retain the CATALOG name field as \$DATA00. Running MODIFY CATALOG with REPLACE VOLUME command on these catalog tables and indexes will fail as it tries to identify the catalog \$DATA00.CATVOL for updating certain records in indexes on catalog tables, such as IXINDE01. Because \$DATA00 has been changed to \$NEW00, the update fails. Hence, consider executing MODIFY LABEL with REPLACE VOLUME command on these catalog objects such that the file labels are updated with the new catalog name. This can be followed by MODIFY CATALOG with REPLACE VOLUME command.

- To minimize unnecessary searching, make your target-spec clause as specific as possible. For example, if SQL objects reside only on subvolumes whose names begin with SQL, specifying \$VOL1.SQL*.* is more efficient than specifying \$VOL1.*.*. The MODIFY LABEL with REPLACE VOLUME command would not have to search for SQL objects on other subvolumes.
- The MODIFY LABEL with REPLACE VOLUME command locks one file label at a time. The file itself is not locked. You must refrain from using the database, including partitions and dependent objects on remote nodes, until the volume names have been modified.
- The MODIFY DICTIONARY utility does not handle:
 - Remote nodes. If you specify a remote file name, the MODIFY LABEL with REPLACE VOLUME command reports a nonfatal error. Each node with dependent objects or partitioned objects must have a version of NonStop SQL/MP that supports MODIFY LABEL with REPLACE VOLUME commands.
 - User-defined SQL object files. For example, MODIFY LABEL with REPLACE VOLUME does not modify a volume name stored in a column of a user-defined table.
 - Volume names stored in SQL object program files.
 - Volume names in Enscribe file labels.
 - Dependent objects. Volume names in the file labels of dependent objects are not modified unless the dependent object is specified in the MODIFY LABEL with REPLACE VOLUME command.

- Partitioned objects. Volume names in the file labels of partitions of tables and indexes are not modified unless those partitions are specified in the MODIFY LABEL with REPLACE VOLUME command.

For a partitioned SQL object, each volume that contains a partition of the object must be specified separately. The MODIFY LABEL with REPLACE VOLUME command does not automatically modify information about all partitions of a partitioned object. You must be aware how the database is distributed and partitioned. Document the MODIFY commands that need to be run, and the nodes they need to be run on, before they are used. Prepare scripts that runs the necessary MODIFY commands.

When you add a new dependent object to the database, update the scripts. While the system is in a consistent state, you can use the DISPLAY USE OF command to locate dependent objects. After MODIFY commands have been run, you can use the VERIFY utility to verify that the database is in a consistent state. While the system is in a consistent state, issue a SELECT from the PARTNS partitions table to locate partitions. Prepare scripts that run the necessary MODIFY commands. After the MODIFY commands are run, use the VERIFY utility to verify that the database is in a consistent state.

- The MODIFY LABEL with REPLACE VOLUME command does not mark SQL object programs as invalid in either the catalog or the object program file label.
- The MODIFY LABEL with REPLACE VOLUME command does not change the redefinition timestamp in either the catalog or the file label.
- DEFINE names can be used in programs to specify the names of catalogs, tables, views, indexes, partitions, and other programs. The current DEFINE set at the time the program is SQL compiled is saved in the object program file. If objects or object programs specified by the DEFINES are moved between the time the program is SQL compiled and the time the program is run, the DEFINES must be changed to reflect the new location of the objects and object programs. This instruction is true regardless of how the database was moved or whether the MODIFY DICTIONARY utility was used to modify the node names and numbers. The MODIFY DICTIONARY utility does not modify the node names in the DEFINE set stored in the object program file.
- After you have modified the DEFINES, if automatic recompilation is enabled, the programs are automatically recompiled using the new DEFINES.
- A catalog and the objects registered in it must be on the same node. However, it is possible for an object to be on a different disk than its catalog. The MODIFY DICTIONARY utility is not intended to correct the situation where a disk containing objects is moved to a new node, but the disk containing the associated catalog is not moved to the new node.

Examples-MODIFY LABEL with REPLACE VOLUME

These examples illustrate the use of the REPLACE clause.

- In this example, the volume name \$DATA is replaced with the volume name \$NEW if the first three characters of the node name are \DA:

```
REPLACE VOLUME $DATA (\DA*) WITH $NEW
```

- In this REPLACE clause, the volume name \$DATA is replaced with the volume name \$NEW if the node name is either \SAM or \IND or if the first three characters of the node name are \DA:

```
REPLACE VOLUME $DATA (\SAM,\IND,\DA*) WITH $NEW
```

- In this REPLACE clause, the volume name \$DATA is replaced with the volume name \$NEW if the node name is anything other than \IND:

```
REPLACE VOLUME $DATA (\* EXCLUDE \IND) WITH $NEW
```

- When you use the CHECKONLY option, note that the amount of information written to the current OUT file depends on whether LISTALL or NO LISTALL is specified. The MODIFY CATALOG command produces this display for each catalog when requested with the LISTALL option:

```
Checking \SYS.$VOL.SUBVOL.T1 label.
```

```
--- \SYS.$VOL.SUBVOL.T1 label requires modification.
```

```
Checking \SYS.$VOL.SUBVOL.T2 label.
```

```
--- \SYS.$VOL.SUBVOL.T2 label does not require modification.
```

- This example on summary information is displayed for CHECKONLY if you specify either LISTALL or NO LISTALL:

```
Summary Information:
```

```
nnn label(s) require modification.
```

```
nnn label(s) do not require modification.
```

- The MODIFY LABEL with REPLACE VOLUME command produces this display when requested with the LISTALL option:

```
Checking \SYS.$VOL.SUBVOL.T1 label.
```

```
--- \SYS.$VOL.SUBVOL.T1 label was modified.
```

```
Checking \SYS.$VOL.SUBVOL.T2 label.
```

```
--- \SYS.$VOL.SUBVOL.T2 label was not modified.
```

- This example on summary information is included whether you specify LISTALL or NO LISTALL:

Summary Information:

nnn label(s) modified.

nnn label(s) not modified.

MODIFY LABEL with REPLACE VOLUME and Partitioned Objects

This example shows the usage of the MODIFY LABEL with REPLACE VOLUME command when you move partitioned objects. Suppose that you have table T1 with one partition at \A.\$DA1.SQL.T1, a second partition at \A.\$DB1.SQL.T1, and a third partition at \C.\$DC1.SQL.T1. The catalogs where the partitions are registered contain references to the other partitions. For example, the catalog where \A.\$DA1.SQL.T1 is registered contains a reference in the PARTNS catalog table to all three partitions, and likewise for the catalogs where \A.\$DB1.SQL.T1 and \C.\$DC1.SQL.T1 are registered.

This SELECT statement illustrates the example:

```
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \A.$DA1.CATSUBV.PARTNS;
FILENAME PARTITIONNAME CATALOGNAME
-----
\A.$DA1.SQL.T1 \A.$DA1.SQL.T1 \A.$DA1.CATSUBV
\A.$DA1.SQL.T1 \A.$DB1.SQL.T1 \A.$DB1.CATSUBV
\A.$DA1.SQL.T1 \C.$DC1.SQL.T1 \C.$DC1.CATSUBV
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \A.$DB1.CATSUBV.PARTNS;
FILENAME PARTITIONNAME CATALOGNAME
-----
\A.$DB1.SQL.T1 \A.$DA1.SQL.T1 \A.$DA1.CATSUBV
\A.$DB1.SQL.T1 \A.$DB1.SQL.T1 \A.$DB1.CATSUBV
\A.$DB1.SQL.T1 \C.$DC1.SQL.T1 \C.$DC1.CATSUBV
>> SELECT FILENAME, PARTITIONNAME, CATALOGNAME
+> FROM \C.$DC1.CATSUBV.PARTNS;
```

FILENAME	PARTITIONNAME	CATALOGNAME
\C.\$DC1.SQL.T1	\A.\$DA1.SQL.T1	\A.\$DA1.CATSUBV
\C.\$DC1.SQL.T1	\A.\$DB1.SQL.T1	\A.\$DB1.CATSUBV
\C.\$DC1.SQL.T1	\C.\$DC1.SQL.T1	\C.\$DC1.CATSUBV

Now suppose that the \$DB1 volume on \A node is renamed to \$XYZ. The MODIFY DICTIONARY commands needed to make the NonStop SQL/MP database consistent after the move are shown:

On node \A, to modify the references to the partition moved from \A.\$DB1 to \$XYZ:

```
>> MODIFY LABEL \A.$DA1.*.*
+> REPLACE VOLUME $DB1 WITH $XYZ;

>> MODIFY LABEL \A.$XYZ.*.*
+> REPLACE VOLUME $DB1 WITH $XYZ;

>> MODIFY CATALOG \A.$DA1.CATSUBV
+> REPLACE VOLUME $DB1 WITH $XYZ;
```

On node \C, to modify the references to the partition moved from \$DB1 to \$XYZ:

```
>> MODIFY LABEL \C.$DC1.*.*
+> REPLACE VOLUME $DB1 WITH $XYZ;

>> MODIFY CATALOG $DC1.CATSUBV
+> REPLACE VOLUME $DB1 WITH $XYZ;
```

MODIFY REGISTER Command

The MODIFY REGISTER command registers a user-defined catalog in the local system catalog.

Each node that uses NonStop SQL/MP has a catalog called the system catalog that contains information about all the catalogs on the node. If a disk containing an SQL database is moved from one node to another, the catalogs that reside on the relocated disk are not automatically registered in the system catalog on the new node. You can access a catalog that is not registered in the system catalog or even create new objects and register them in such a catalog. However, to make the system consistent, you should register all user-defined catalogs in the SQL system catalog.

```
MODIFY [ DICTIONARY ] REGISTER target-spec

    [ [,] option ] ...;

target-spec is:

    CATALOG catalog-list-1 [ EXCLUDE catalog-list-2 ]

catalog-list-n (where n = 1,2) is:

    { catalogset
      ( catalogset [ , catalogset ] ... ) }

option is:

    [ | ALLOWERRORS [ OFF | ON | number-of-errors ] | ]
    [ | [ NO ] LISTALL | ]
```

CATALOG *catalog-list-1* [EXCLUDE *catalog-list-2*]

identifies one or more SQL catalogs to be registered in the system catalog. If the catalog is already registered in the system catalog, a warning is reported, and the command continues.

The optional EXCLUDE *catalog-list-2* clause specifies catalogs to be excluded from *catalog-list-1*.

catalogset specifies one catalog or a set of catalogs. To specify a single catalog, enter the name of the catalog (the name of the subvolume that contains the catalog). To specify multiple volumes, use wild-card characters. You can use these wild-card characters:

- * Matches 0 to 8 characters in the position where it appears. Specifying only an asterisk indicates any name is acceptable. To specify all catalogs, use either \$*.* or *.*.
- ? Matches any single character.

For example, \$DATA.* specifies all catalogs on the volume \$DATA, while *.* specifies all catalogs on the node. *VOL* matches NEWVOL, OLDVOL1, and VOL45. VOL? matches VOL1 and VOLX but not VOL or VOL48.

The MODIFY REGISTER command functions requires that a subvolume contains a valid catalog if the subvolume contains the catalog table TABLES (file code must be 581).

ALLOWERRORS [OFF | ON | *number-of-errors*]

determines handling of nonfatal errors. MODIFY REGISTER reports two classes of errors: fatal errors and nonfatal errors. The MODIFY command always terminates after reporting a fatal error. Nonfatal errors are handled depending on the value of the ALLOWERRORS option:

OFF	The MODIFY REGISTER command terminates immediately after the first nonfatal error is encountered.
ON	The MODIFY REGISTER command continues, regardless how many nonfatal errors are encountered.
<i>number-of-errors</i>	The MODIFY REGISTER command continues until the number of nonfatal errors exceeds the value of <i>number-of-errors</i> .

If you specify ALLOWERRORS without ON, OFF, or the number of errors, ALLOWERRORS ON is the default.

If you do not specify ALLOWERRORS, ALLOWERRORS OFF is the default.

When MODIFY REGISTER continues processing after a nonfatal error has occurred, it advances to the next SQL catalog.

Each MODIFY REGISTER error describes whether the error is fatal or nonfatal.

[NO] LISTALL

specifies how much information MODIFY REGISTER writes to the current OUT file. If LISTALL is specified, MODIFY REGISTER reports the name of each SQL catalog registered. If NO LISTALL is specified, only summary information is reported.

LISTALL is the default.

Considerations—MODIFY REGISTER

- You must be logged on as the super ID to run a MODIFY DICTIONARY command, unless you specify the CHECKONLY option.
- NonStop SQL/MP uses the TMF subsystem to protect the integrity of the database during the MODIFY REGISTER operation. MODIFY commands are not allowed inside a user-defined transaction.
- The MODIFY REGISTER command is one of a set of commands that uses the MODIFY DICTIONARY utility. The other related commands are MODIFY LABEL with REPLACE NODENUMBER (to change node numbers in file labels) and MODIFY CATALOG (to change node names in SQL catalogs).
- MODIFY REGISTER does not mark SQL object programs as invalid in either the catalog or in the object program file label.
- MODIFY REGISTER does not change the redefinition timestamp in either the catalog or the file label.
- The MODIFY DICTIONARY utility does not handle remote nodes. If you specify a remote catalog name, the MODIFY REGISTER command reports a nonfatal error. Because of this, each node with dependent objects or partitioned objects must have a version of NonStop SQL/MP that supports MODIFY commands.
- The MODIFY REGISTER command registers a catalog in the system catalog. However, there is no command that removes information about a catalog from the system catalog. For example, suppose that a disk containing the catalog \$VOL1.CAT and the objects registered in it is moved from \SYSA to \SYSB. The MODIFY DICTIONARY REGISTER option can be used to register the catalog in the system catalog on \SYSB, but a licensed SQLCI2 process must be used to remove the information about the \$VOL1.CAT catalog from the system catalog on \SYSA.
- A catalog and the objects registered in it must be on the same node. However, it is possible for an object to be on a different disk than its catalog. The MODIFY DICTIONARY utility is not intended to correct the situation where a disk containing objects is moved to a new node but the disk containing the associated catalog is not moved to the new node.

Examples—MODIFY REGISTER

- This example shows what is displayed for each catalog, if LISTALL is specified:

```
Registering catalog \SYS.$VOL.CAT1.
--- \SYS.$VOL.CAT1 was registered.
Registering catalog \SYS.$VOL.CAT2.
--- \SYS.$VOL.CAT2 was not registered.
```

- This summary information is displayed if either LISTALL or NO LISTALL is specified:

```
Summary Information:
  nnn catalog(s) registered.
  nnn catalog(s) not registered.
```

- For a comprehensive example, see the [MODIFY LABEL](#) on page M-20.

Multibyte Character Sets

SQL supports two multibyte character sets:

- Kanji
- KSC5601

Multibyte character sets are described under the entry Character Sets and can be associated with columns, literals, host variables, and parameters. (You cannot use multibyte character sets in collations. SQL always collates multibyte character values according to the binary representation of the characters.)

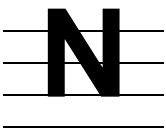
System Default National Character Set

Each node in a network that runs NonStop SQL/MP has a system default national character set associated with it. SQL uses the system default national character set when your SQL statements specify the data type NATIONAL CHARACTER or NCHAR, or when you use the national character form of a string literal.

The released system default national character set is Kanji, but your site can change the default to one of the other multibyte character sets during a SYSGEN. You can use the system procedure MBCS_DEFAULTCHARSET (described in the *Guardian Procedure Calls Reference Manual*) to determine the current system default national character set for a node.

SQL returns an error if you try to create an SQL column with a NATIONAL CHARACTER or NCHAR data type on a node with a system default multibyte character set that SQL does not support. The same error occurs if you use a string literal with the prefix N (indicating the system default multibyte character set) on such a node.

If you run SQL DDL or DML statements that use the national character data type to create tables or manipulate data on a node with a different system default multibyte character set, the character set used is the default on the node that runs the command, not the node on which the tables reside.



NAME Command

NAME is an SQLCI report writer command that assigns an alias to a column in the select list of the SELECT command. You can then use the alias to refer to the column in any other part of your report definition.

NAME is convenient for defining abbreviations for long column names or for assigning informative names to columns that consist of expressions.

```
NAME column alias ;
```

column

identifies a column in the select list of the SELECT command. It can be a column name, an alias, or COL *number* (which specifies the position of the column in the select list). It cannot be a detail alias.

alias

is an SQL identifier that is unique among column names in the select list and among existing aliases. It becomes the alias for the specified column.

Consideration—NAME Command

If you specify an alias, it becomes the default heading for the column. If you specify more than one alias for the same column, the most recently defined alias is the default heading.

Example—NAME Command

This example defines an alias for the second column in a select list. The output shows the effect of the alias on the heading.

```
>> SET LIST_COUNT 0;
>> SELECT EMPNUM, SALARY/12 FROM PERSONL.EMPLOYEE;
S> NAME COL 2 MONTHSAL;
S> DETAIL EMPNUM, MONTHSAL;
S> TOTAL MONTHSAL;
S> LIST FIRST 1;
EMPNUM  MONTHSAL
-----
      1      14625.000000000000
```

NAME Option

The NAME option specifies an operation name for an operation started by a statement that includes the NAME option. Use the operation name in subsequent CONTINUE statements or to identify EMS messages sent by the operation. For more information, see [REPORT Option](#) on page R-3.

NAME *operation-name*

operation-name

is an SQL identifier to be the name for the operation. *operation-name* should normally be unique on the node so that messages from the operation are not confused with messages for other operations using the same name. Uniqueness is not required for the operation to work correctly. If you omit the NAME option, the name of the operation is the first two words of the statement that initiated the operation concatenated by an underscore (for example, ALTER_TABLE).

Consideration—NAME Option

The operation name appears in EMS messages as the token ZSQL-TKN-OP-TYPE and ZAUD-TKN-OP-TYPE. For more information about EMS messages sent by NonStop SQL/MP, see the *SQL/MP Messages Manual*.

Example—NAME Option

This example CREATE INDEX statement uses the NAME option to name the index-creation operation CREATE_INDADV:

```
CREATE INDEX INDADV ON STUDENTS (ADVISOR,CLASS)
  WITH SHARED ACCESS NAME CREATE_INDADV COMMIT BY REQUEST;
```

Name Resolution

Name resolution is the mapping of a name in an SQL statement to a particular table, view, index, program, partition, collation, catalog, or EDIT file. Name resolution includes mapping DEFINES to physical names and fully qualifying partially qualified physical names using the current default node, volume, subvolume, and catalog names.

The time at which name resolution occurs depends upon the statement and upon whether a CONTROL QUERY BIND NAMES AT EXECUTION directive was in effect at the time the statement was compiled or prepared (compiled by executing a PREPARE or EXECUTE IMMEDIATE statement). Names in an INVOKE statement in a host language program are always resolved during host language compilation, along with names in host language statements that are not SQL statements.

By default, SQL resolves names in a static SQL statement at program startup, resolves names in a prepared statement at the time the PREPARE or EXECUTE IMMEDIATE executes, and resolves names in a non-prepared SQLCI statement at the time you enter the statement. However, if a CONTROL QUERY BIND NAMES AT EXECUTION directive is in effect at a statement's compilation, preparation, or entry (for static, prepared, or SQLCI statements, respectively), then SQL resolves names in the statement at the time the statement executes instead.

For more information about changing the time at which names are resolved, see [CONTROL QUERY Directive](#) on page C-74. For information about the resolution of DEFINE names, see [DEFINES](#) on page D-27. For information about name resolution in host programs and the relationship between name resolution and various compilation options, see the *SQL/MP Programming Manual for COBOL* or the *SQL/MP Programming Manual for C*.

Names

NonStop SQL/MP uses six main types of names:

- SQL identifiers
- Guardian names
- OSS names
- Host identifiers
- DEFINE names
- Catalog names

Rules for SQL identifiers, host identifiers, Guardian names, and OSS names are described in separate entries for those topics. Rules for DEFINE names and catalog names are described in the entries for [DEFINES](#) on page D-27, and [Catalogs](#) on page C-8.

Rules for names of other entities used in NonStop SQL/MP are described in terms of these six types of names. For example, a table name is a Guardian name, the name of a prepared statement is an SQL identifier, and so forth.

Generally, the rules for naming an entity used by NonStop SQL/MP are described in the main entry that describes that entity. For example, the rules for table names are described in [Tables](#) on page T-1, and the rules for naming prepared statements are described in [PREPARE Statement](#) on page P-25.

Case is generally not significant in SQL/MP names, although it is significant in names of host variables in the C programming language, in string literals, and in the c89 command.

For more information, see the entry for a specific type of name.

NEWLINE_CHAR Option

NEWLINE_CHAR is an option of the SQLCI report writer SET STYLE command that specifies the character that indicates a new line in a column heading.

NEWLINE_CHAR " <i>character</i> "

character

is a single-byte character to mark the end of a line in a heading string. The default is "/".

Consideration—NEWLINE_CHAR

For information about how to create headings, see [DETAIL Command](#) on page D-47.

Example—NEWLINE_CHAR

This example sets the new-line character to an exclamation point, then uses it in a DETAIL command to create a two-line heading:

```
>> SET STYLE NEWLINE_CHAR "!";
S> DETAIL EMPNUM HEADING "Employee!Number" CENTER, ...
S> LIST FIRST 1;
Employee
  Number
-----
234
```

Nonaudited Tables

Nonaudited tables are tables that are not audited by TMF, the main functional component of the TMF product. TMF recovery operations do not protect nonaudited tables from node failure or media failure.

NonStop SQL/MP creates audited tables by default but you can specify the creation of a nonaudited table (or change an audited table to a nonaudited table) using the AUDIT file attribute for the table.

For more information, see [AUDIT File Attribute](#) on page A-74 or [TMF Transactions](#) on page T-6.

NOPURGEUNTIL File Attribute

NOPURGEUNTIL is a Guardian file attribute that specifies an expiration date and time after which a table or index can be purged or dropped. NOPURGEUNTIL applies to key-sequenced, relative, and entry-sequenced tables and to indexes.

SQL stores the date and time in local civil time (LCT).

NOPURGEUNTIL	{ mmmbddbbyyyy [, hh:nn] }
	{ ddbmmmbbyyyy [, hh:nn] }
	{ hh:nn }

b is required space

mmm is a 3-character month value (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)

dd is a 2-digit day value (01, 02, ... , 31)

yyyy is a 4-digit year value

hh is a 2-digit hour value (00, 01, ... , 23)

nn is a 2-digit minute value (00, 01, ..., 59)

Defaults

The default is NOPURGEUNTIL 0, which specifies that the object can be purged at any time.

If you specify a date but omit time, the time 00:00 is used.

If you specify a time with no date, the current date is used.

Example—NOPURGEUNTIL

These NOPURGEUNTIL values prevent purging until 2003:

```
NOPURGEUNTIL JAN 01 2003
NOPURGEUNTIL 31 DEC 2003, 23:59
```

NULL Predicate

NULL is a predicate that determines whether a column contains a null value.

```
row-value-specification IS [ NOT ] NULL
```

```
row-value-specification is:
```

```
{ expression [, expression ] ... }
{ ( expression [, expression ] ... ) }
```

Considerations—NULL

- If any expression in the NULL predicate evaluates to a value other than null, the IS NOT NULL predicate evaluates to TRUE; otherwise, IS NOT NULL evaluates to FALSE.

This chart summarizes expression evaluation for null predicates. *rvs* stands for row value specification. Degree is the number of expressions in row value specification.

Expression	<i>rvs</i> is NULL	<i>rvs</i> is NOT NULL	NOT <i>rvs</i> is NULL	NOT <i>rvs</i> is NOT NULL
degree 1: null	TRUE	FALSE	FALSE	TRUE
degree 1: not null	FALSE	TRUE	TRUE	FALSE
degree>1: all null	TRUE	FALSE	FALSE	TRUE
degree>1: some null	FALSE	FALSE	TRUE	TRUE
degree>1: none null	FALSE	TRUE	TRUE	FALSE

Note that the expression

```
row-value-specification IS NOT NULL
```

is not equivalent to the expression

```
NOT ( row-value-specification IS NULL )
```

- If all the expressions in the NULL predicate evaluate to null, the IS NULL predicate evaluates to TRUE; otherwise, IS NULL evaluates to FALSE.

Examples—NULL

- This example finds all rows with a null value in the SALARY column:

```
SALARY IS NULL
```

- This example evaluates to true if the expression (PRICE + TAX) evaluates to null:

```
(PRICE + TAX) IS NULL
```

- This example evaluates to true if the value in :JOBCODE is not null:

```
:JOBCODE IS NOT NULL
```

- This example finds all rows where both FIRST_NAME and SALARY have a null value:

```
FIRST_NAME, SALARY IS NULL
```

Null Values

A null value is a special symbol, independent of data type, that represents an unknown or inapplicable value. A null value indicates that an item has no value. For sorting purposes, SQL considers null values greater than all other values.

You cannot store a null value in a column, either with INSERT or UPDATE, unless the column was declared to allow null values when it was created.

Any row of a column that allows null values can be empty. In SQL, a column that allows null values has two extra bytes associated with it in each row. A -1 stored in those two bytes indicates that the column has a null value for that row; a 0 indicates a null value.

Using Null Values Versus Default Values

Various scenarios exist in which a row in a table might contain no value for a specific column. For example:

- A database of telemarketing contacts might have AGE fields empty if contacts did not give their age.
- An order record might have a DATE_SHIPPED column empty until the order is actually shipped.
- An employee record for an international employee might not have a social security number.

You allow null values in a column when you want to convey that a value in the column is either unknown (such as the age of a telemarketing contact) or not applicable (such as the social security number of an international employee).

In deciding whether to allow nulls or use defaults, also note these points:

- Null values are not the same as blanks. Two blanks can be compared and found equal, while the equivalence of two null values is indeterminate.
- Null values are not the same as zeros. Zeros can participate in arithmetic operations, while null values are excluded from arithmetic.

Defining Columns That Allow or Prohibit Nulls

CREATE TABLE and ALTER TABLE define all the column attributes for columns of tables. You use these statements to specify whether a new column allows null values.

A column allows null values unless the column definition includes the NOT NULL clause or the column is part of the primary key of the table.

A null value is also the default value for a column unless the column definition includes either the DEFAULT (excluding DEFAULT NULL) or the NO DEFAULT clause. (The default value for a column is the value SQL inserts in a row when an INSERT statement omits a value for a particular column or when a column is added to an existing table.)

These sample column definitions allow or prohibit null values as indicated:

CA	INTEGER		Allows nulls, default null
CB	INTEGER	DEFAULT SYSTEM	Allows nulls
CC	INTEGER	NO DEFAULT	Allows nulls
CD	INTEGER	DEFAULT SYSTEM NOT NULL	Prohibits nulls
CF	INTEGER	DEFAULT NULL	Allows nulls, default null

The *SQL/MP Installation and Management Guide* discusses defining columns with the NULL and DEFAULT clauses in detail.

Determining Whether a Column Allows Nulls

To determine whether a column accepts null values, you can query the COLUMNS catalog table or you can use INVOKE to list the table description in SQL format (the default format from SQLCI) and check the column definitions. The COLUMNS table contains descriptions of all columns of all tables registered in a catalog (as recorded in the TABLES catalog table). The one-character NULLALLOWED column contains a Y if a null value is allowed, and an N if not.

These examples illustrate how to display information through SQLCI about whether columns allow or prohibit null values:

- This example queries the value of the NULLALLOWED column in the COLUMNS catalog table for the description of a particular column in a particular table. The

example uses the LIKE predicate to avoid entering the whole, exact table name for the OD2 table. The column of interest is the DELIV_DATE column in table OD2.

```
>> SELECT NULLALLOWED FROM COLUMNS
+>     WHERE TABLENAME LIKE "%OD2%" AND
+>     COLNAME = "DELIV_DATE";
NULLALLOWED
-----
Y
--- 1 row(s) selected.
```

- This example queries the COLUMNS catalog table to display the value for the NULLALLOWED column for all the columns of a particular table:

```
>> SELECT TABLENAME, COLNAME, NULLALLOWED
+>     FROM COLUMNS
+>     WHERE TABLENAME LIKE "%OD2%";
```

- This example invokes a table description in SQL format (the default format through SQLCI). The display shows NOT NULL for columns whose definition prohibits null values.

```
>> INVOKE OD2;
-- Definition of table \SYS1.$VOL1.SALES.OD2
-- Definition current at 16:36:57 - 05/23/89
(
ORDERITEM      DECIMAL( 6, 0 ) UNSIGNED NO DEFAULT
                NOT NULL
, ORDERNUM      NUMERIC( 6, 0 ) UNSIGNED NO DEFAULT
                NOT NULL
, ORDER_DATE    NUMERIC( 6, 0 ) NO DEFAULT
, DELIV_DATE    NUMERIC( 6, 0 ) NO DEFAULT
, SALES_REP     DECIMAL( 4, 0 ) UNSIGNED DEFAULT SYSTEM
, CUSTNUM       DECIMAL( 4, 0 ) UNSIGNED NO DEFAULT
)
```

Specifying Null Values in Host Programs

Host programs use indicator variables to indicate the presence of null values. For more information, see [Indicator Variables and Indicator Parameters](#) on page I-11 or the SQL/MP programming manual for your host language.

DISTINCT, GROUP BY, and ORDER BY With Null Values

In evaluating the DISTINCT, GROUP BY, and ORDER BY clauses, SQL considers all null values to be equal. Additional considerations for these clauses are:

DISTINCT Null values are considered duplicates; a result has at most one null

GROUP BY The result has at most one null group

ORDER BY Null values are considered greater than nonnull values

Null Values and Expression Evaluation

This chart summarizes the results of expression evaluation with null values.

Expression Type	Condition	Result
Boolean (AND, OR, NOT)	Either value null	True, false, or null See truth tables in Search Conditions on page S-5
Arithmetic	Either or both values null	Null
NULL predicate		See SEARCH CONDITION
Aggregate functions (except COUNT)	Evaluated after eliminating nulls	Null if set is empty
COUNT COUNT DISTINCT	Evaluated after eliminating nulls	Zero if set is empty
Comparison: > < = >= <= <> LIKE	Either value null	Null
IN predicate	Expression is null	Null
Subquery	No values returned	Null

NULL_DISPLAY Option

NULL_DISPLAY is an option of the SQLCI report writer SET STYLE command that defines a character to represent NULL print items in a report.

```
NULL_DISPLAY "character"
```

character

is a printable, single-byte character used to represent the null value in a printed report. The default is ? (question mark).

Example—NULL_DISPLAY

This example adds a column with null values and prints the column twice, using a different value for the NULL_DISPLAY option each time:

```
>> ALTER TABLE PRJ ADD COLUMN DEPT PIC X(6) DEFAULT NULL;
--- SQL operation complete.
>> SELECT DEPT FROM PRJ;
DEPT
-----
?
?
?
--- 3 row(s) selected.
>> SET STYLE NULL_DISPLAY "Z";
```

```
>> SELECT DEPT FROM PRJ;
DEPT
-----
Z
Z
Z
```

Numeric Data Types

[Table N-1](#) lists the numeric data types available in NonStop SQL/MP. A numeric data type is compatible with any other numeric data type, but not with character, date-time, or interval data types.

Table N-1. Numeric Data Types in SQL—Binary Types

SQL Designation	Description	Size or Range (1)
NUMERIC(1,s) to NUMERIC(18,2)	Exact binary number with optional scale; signed or unsigned for 1 to 9 digits; signed required for 10 or more digits	1 to 18 digits stored: 1 to 4 digits in 2 bytes 5 to 9 digits in 4 bytes 10 to 18 digits in 8 bytes
PIC S9V9 COMP to PIC S9(18) COMP	Binary number; same as NUMERIC	1 to 18 digits; stored as NUMERIC
SMALLINT	Binary integer; signed or unsigned	–32768 to +32767 or 0 to 65535; stored in 2 bytes
INTEGER	Binary integer; signed or unsigned	–2147483648 to +2147483647 or 0 to 4294967295; stored in 4 bytes
LARGEINT	Binary integer; signed only	–2**63 to 2**63-1; stored in 8 bytes

Table N-2. Numeric Data Types in SQL—Floating Point Types

SQL Designation	Description	Size or Range (1)
FLOAT [(pre)]	Approximate floating point number; pre-designates from 1 through 54 bits of precision	+/-8.62 times 10**-78 through +/-1.16 times 10**77; stored: pre 1 to 22 in 4 bytes, pre 23 to 54 in 8 bytes
REAL	Approximate floating point number (22 bits)	Approximately 7 decimal digits of precision; same range as FLOAT; stored in 4 bytes
DOUBLE PRECISION	Approximate floating point number (54 bits)	Approximately 16 decimal digits of precision; same range as FLOAT; stored in 8 bytes

Table N-3. Numeric Data Types in SQL—Decimal Types

SQL Designation	Description	Size or Range (1)
DECIMAL (1,s) to DECIMAL (18,s) and PIC S9V9 DISPLAY to PIC S9(18) DISPLAY	Decimal number with optional scale; stored in ASCII; must be signed if 10 or more digits, otherwise can be signed or unsigned.	1 to 18 digits; byte length equals the number of digits

Considerations—Numeric Data Types

- All the preceding data types are exact data types except for the floating point types, which are approximate data types. Exact data types have greater precision. Approximate data types are subject to rounding error and should not be used for equality comparisons or other operations that require exact results.
- Floating point (approximate) data types should be used for very large or very small numbers that cannot be stored in other data types. If you can represent column values with an exact data type, use the exact data type instead of a floating point data type.
- For more information about numeric data types, see [Data Types](#) on page D-1.

Numeric Literals

A numeric literal represents a numeric value. Each numeric literal has the data type NUMERIC and the minimum precision required to represent the value it specifies.

A simple numeric literal (one without an exponent) can include up to 18 digits (0 through 9), a plus sign (+) or a minus sign (–), and a period (.) that indicates a decimal point. Leading zeros do not count toward the 18-digit limit; trailing zeros do.

A sign in a simple numeric literal must be the first character of the numeric literal. A numeric literal without a sign is considered to be a positive number.

A simple numeric literal that does not include a decimal point is considered to be an integer.

A numeric literal in scientific notation is a simple numeric literal followed by an exponent expressed as the letter *E* or *e* followed by an optionally signed integer.

Numeric values expressed in scientific notation are handled as type REAL if they include no more than seven digits before the exponent, but handled as type DOUBLE PRECISION if they include eight or more digits. Because of this, trailing zeros after a decimal can sometimes increase the precision of a numeric literal used as a DOUBLE PRECISION value. For example, if XYZ is a table that consists of one DOUBLE PRECISION column:

```
INSERT INTO XYZ VALUES (1.000000000E-10);
```

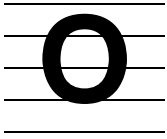
is more precision than

```
INSERT INTO XYZ VALUES (1.0E-10);
```

Example—Numeric Literals

These are all numeric literals:

477	580.45	+005	–.3175	1300000000
99.	–0.123456789012345678	99E–2		



OBEY Command

OBEY is an SQLCI command that executes SQL statements and SQLCI commands from a file.

OBEY executes the statements and commands exactly as if you had entered them from the terminal. After execution, SQLCI closes the file but does not return any setting changed by the commands (such as a session attribute) to a previous state.

OBEY is often used to set DEFINES or define reports, but is useful in any situation in which you repeat a sequence of statements or commands. Using OBEY to execute statements and commands from files makes tedious jobs faster, easier to reproduce, and more reliable.

```
O[BEY] cmd-file [ ( section [ , section ] ... ) ] ;
```

cmd-file

is the name of a closed file that contains commands to execute.

The file (called a *command file* or an *OBEY command file*) is usually an EDIT disk file, but can also be a device or process. It cannot be the SQLCI IN, OUT, or log file, or an executing command file, however, because these files are open.

section

is the name of a section in the file to execute.

For each *section* you specify, SQLCI executes the lines in the file from the named section header to the next section header (or the end of the file). If you specify more than one section, SQLCI executes the sections in the order in which they appear in the file, not in the order you specify them. If more than one section in the file has the name you specify, SQLCI executes only the first one; other sections with the same name are ignored.

If you omit *section*, SQLCI executes all lines in the file.

Considerations—OBEY

- You can use named parameters as literals in DML statements or SQLCI commands within command files. Use SET PARAM to supply values for the parameters before you use OBEY to execute the statements or commands. For more information, see [Parameters](#) on page P-11.

- Specify sections within a command file by including a section header starting in column 1 at the beginning of each section:

?SECTION *section-name*

The *section-name* is an SQL identifier that is the name of the section. Each section name within a file should be unique, because SQLCI executes only the first section it finds that has the name you specify in an OBEY command.

- Most command files are simply EDIT files that contain SQLCI commands (and, optionally, section headers). You can create or modify command files any way you normally create or modify EDIT files, typically with the EDIT or TEDIT text editor.

NonStop SQL/MP also creates three types of command files for you:

- SQLCOMP creates a command file that sets DEFINES used by a program if you compile the program with the EXPLAIN DEFINES option and specify OBEYFORM. See the SQL/MP programming manual for your host language.
- The SAVE command creates a command file that sets up a report definition or other SQLCI session options. For more information, see [SAVE Command](#) on page S-2 or the *SQL/MP Report Writer Guide*.
- The LOG COMMANDS command logs SQLCI commands you enter. You can then create an SQLCI OBEY command file from the log file.

In all three cases, you can use the command file as it is or modify it with an editor.

- You can nest command files to four levels beyond the SQLCI IN file. For example, if you enter OBEY FILE1 at the terminal (IN file) and FILE1 contains OBEY FILE2, FILE2 contains OBEY FILE3, and FILE3 contains OBEY FILE4, FILE4 cannot contain an OBEY command. You can have at most five files open including the IN file.

A command file cannot include an OBEY command that executes commands from the same command file, however, even if the commands are within another section of the file. (Inclusion would violate the restriction that the command file specified in the OBEY must be closed, because a command file remains open while SQLCI executes it.)

Within a command file, SQLCI executes commands until it reaches the end of a section, the end of a file, another OBEY, or an EXIT command. When it reaches the end of a section or file, SQLCI returns to the line following the OBEY command that initiated execution of that section or file.

- If the BREAK_KEY option is ON, you can stop the execution of commands in a command file by pressing the Break key at the terminal from which you issued the OBEY. SQLCI closes the command file and prompts you for a new command. If a transaction is in progress, it is rolled back.

If BREAK_KEY is OFF, pressing the Break key interrupts execution and passes control to the previous Break key owner, usually TACL. From TACL, enter PAUSE to resume or STOP to terminate SQLCI.

Examples—OBEY

- This example shows the contents of a simple command file that sets DEFINE values:

```
SET DEFMODE ON;
ADD DEFINE =REP, FILE \SYS1.$VOL2.SALES.SALESREP;
ADD DEFINE =CUST, FILE \SYS1.$VOL2.SALES.CUSTOMER;
ADD DEFINE =ORD, FILE \SYS1.$VOL2.SALES.ORDERS;
```

If the commands are in a file named DEFS, you can execute them from SQLCI by typing,

```
OBEY DEFS;
```

- This example shows a command file that uses both sections and parameters. The example also shows the SQLCI output when OBEY executes a short section from a command file.

The contents of UPDOP the command file are:

```
?SECTION NEWJOB
UPDATE PERSNL.EMPLOYEE
    SET JOBCODE=?JCOD WHERE EMPNUM=?EMPN;
?SECTION NEWSAL
...
?SECTION NEWDEPT
...
```

- This example executes section NEWJOB from the command file. The first two commands (SET PARAM and OBEY) are entered by the user; everything else is printed by SQLCI, echoing and responding to commands from the file as it executes them.

```
>>SET PARAM ?EMPN 557, ?JCOD 500;
>>OBEY UPDOP (NEWJOB);
>>?SECTION NEWJOB
>>UPDATE PERSNL.EMPLOYEE
+> SET JOBCODE = ?JCOD WHERE EMPNUM = ?EMPN;
--- 1 row(s) updated.
```

OCTET_LENGTH Function

The OCTET_LENGTH function returns the length of a character string in bytes.

OCTET_LENGTH (*character-string*)

where *character-string* is:

```
{string-literal      }
{column-name         }
{param-name          }
{host-var-name       }
{UPSHIFT function    }
{character-expression}
```

character-string

specifies the string for which the length is to be returned.

Considerations—OCTET LENGTH Function

- SQL returns the result as a two-byte signed integer with a scale of zero.
- If *character-string* is a null value, SQL returns a length of null.
- For a column declared as a fixed CHAR, SQL returns the maximum length of that column. For a VARCHAR column, SQL returns the actual length of the string stored in that column.
- The OCTET_LENGTH and CHAR_LENGTH functions are similar. The CHAR_LENGTH function returns the number of characters in the string. The result of both functions is the same for single-byte character data types. For multibyte character data types, the two functions return different results.

Examples—OCTET LENGTH Function

- This example returns the value 6:

```
OCTET_LENGTH ( "Robert" )
```

- This example returns the value 6:

```
OCTET_LENGTH ( _KANJI "abcdef" )
```

OPEN Statement

OPEN is a DML statement that opens a cursor in a host program. OPEN executes the SELECT associated with the cursor, positions the cursor before the first row selected, and returns statistics to the SQLSA.

In dynamic SQL, OPEN also specifies parameters for the SELECT.

```
OPEN { cursor } [ USING :var [, :var ]... ]
      { :cursor-var } [ USING DESCRIPTOR :in-sqllda ]
```

cursor

is the name of a cursor defined by DECLARE CURSOR.

:cursor-var

(dynamic SQL only) is the name of a host variable of SQL type CHAR or VARCHAR that contains the name of a cursor defined by DECLARE CURSOR.

USING *:var* [, *:var*]...

(dynamic SQL only) specifies host variables that contain values for parameters used in a FETCH command for the cursor. Use this clause when you know the descriptions of parameters in the prepared SELECT.

USING DESCRIPTOR *:in-sqllda*

(dynamic SQL only) specifies an SQLDA filled by DESCRIBE INPUT that points to values for parameters used in a FETCH for the cursor. Use this clause when you do not know the descriptions of parameters in the prepared SELECT.

Considerations—OPEN

- To execute OPEN, you must have read authority for tables or protection views referred to in the SELECT associated with the cursor. If the cursor refers to a shorthand view, you must have read authority for tables or protection views underlying the shorthand view. If the cursor was declared FOR UPDATE, you must also have write authority to the tables.
- If a cursor is declared on audited tables or protection views and acquires locks, a TMF transaction must be in progress when you open the cursor. The OPEN statement associates the cursor with the transaction.

OPEN itself acquires no locks unless a sort operation must order selected rows. (A FETCH in the cursor acquires locks unless you specify BROWSE access in the SELECT and subqueries.)

- To use a cursor, you must first declare it with `DECLARE CURSOR` and then open it with `OPEN`. After a successful `OPEN`, you use `FETCH` to retrieve data.

You cannot open a cursor that is already open in the program. (Use `CLOSE` or `FREE RESOURCES` to close a cursor before program termination.)

Example—OPEN

This example declares and opens a cursor, uses `FETCH` to retrieve data, then closes the cursor:

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR
  SELECT COL1, COL2, COL3 FROM =PARTS
  WHERE COL1 >= :HOSTVAR1 ORDER BY COL1 BROWSE ACCESS;
EXEC SQL OPEN CURSOR1;
EXEC SQL FETCH CURSOR1 INTO :HOSTVAR1, :HOSTVAR2, :HOSTVAR3;
EXEC SQL CLOSE CURSOR1;
```

OSS NAMES

OSS names are names used for files that belong to the Open System Services environment on a NonStop System, rather than to the Guardian environment.

NonStop SQL/MP databases reside in the Guardian environment of a NonStop system but you can access NonStop SQL/MP databases with programs from either the Guardian environment or the OSS environment. NonStop SQL/MP program files from the OSS environment have OSS names.

OSS names have two forms, pathnames and ZYQ names.

A pathname is a standard form of OSS file name and is described in detail in documentation for the OSS environment. You use pathnames to identify files (including SQL program files) within the OSS environment. Each pathname can have up to 1023 characters and a typical pathname might look like this:

```
/a/b/c/d/myfile
```

You use a pathname to specify an SQL program in an OSS file when you invoke `c89` to compile an SQL program, but you cannot use a pathname as a parameter on an SQL statement or in an `SQLCI` command.

Each pathname is associated with a physical file that can have other pathnames as well and that also has a special form of Guardian name referred to as a ZYQ name. A ZYQ name is so-called because the subvolume portion of the name always begins with the letters ZYQ. The full form of the name is:

```
$vol.ZYQnnnnn.Ziiiiiii
```

`nnnnn` and `iiiiiii` are alphanumeric strings that identify the file within the file system. Each ZYQ file is an OSS file that has one or more corresponding pathnames.

Each SQL program in an OSS file has at least one pathname and exactly one ZYQ name. You use the pathname to identify the file in the OSS environment where you

create, compile, execute, and maintain the program. NonStop SQL/MP uses the ZYQ name to identify the file in an SQL/MP catalog.

To determine a pathname from a ZYQ name, use the DETAIL option on the FILEINFO or FUP INFO command. Both these commands return one of the pathnames of an OSS file as part of the DETAIL display.

OUT Command

OUT is an SQLCI command that directs SQLCI output to a specific file or closes the current OUT file and redirects output to the initial OUT file.

```
OUT [ list-file [ CLEAR ] ] ;
```

list-file

is the name of the file to which you want the output written. For *list-file*, specify a device, disk, or process file. The OUT file cannot be the same file as any other current output file such as the LOG, INVOKE TO, or OUT_REPORT file unless the file is a terminal or process. The OUT file can be the same as the IN file.

If you omit *list-file*, SQLCI closes the current OUT file and sends output to the file that was the OUT file at the beginning of the SQLCI session (usually your terminal).

If you specify a nonexistent disk file as *list-file*, SQLCI creates an EDIT file of that name.

CLEAR

clears the new OUT file before anything is written to it. If you omit this option, information is appended to the file. CLEAR is ignored unless the file is a disk or process file.

Consideration—OUT

SQLCI writes all information that it produces to the OUT file, including output from commands such as SHOW and DISPLAY STATISTICS, data from SELECT commands (unless you specify an OUT_REPORT file), and diagnostic messages. SQLCI also writes the final version of any command you fix with FC command to the OUT file, but not the characters you enter while fixing it.

If the IN file is the same as the OUT file (which is the case in interactive usage), SQLCI also writes the current prompt and lines that you enter to the OUT file.

SQLCI ejects a page before and after each report from a SELECT command. If the OUT file is a spooler file and you do not specify an OUT_REPORT file, each report appears on a separate page (or pages) and does not include the SELECT command or diagnostic messages.

Example—OUT

In this example, SQLCI output is directed to a printer. After the SELECT command is executed, the output is redirected to the initial OUT file.

```
>> OUT $$.#FASTPRT;
>> SELECT * FROM EMPLOYEE;
>> OUT;
```

OUT_REPORT COMMAND

OUT_REPORT is an SQLCI report writer command that directs the formatted output of a SELECT command to a specified report file, instead of to the OUT file.

```
OUT_REPORT [ file ]
[ CLEAR                                     ] ;
[ SPOOL3 spool-option [, spool-option] ... ]

spool-option is:
{ LOC loc-name           }
{ FORM form-name        }
{ REPORT rpt-name       }
{ COPIES number        }
{ PAGESIZE number      }
```

file

is a Guardian name that specifies a disk, device, or process file (such as a spooler collector) for reports. *file* cannot be the current IN file or any current output file (such as the LOG or OUT file) unless it is a terminal or process.

If you specify the name of a nonexistent disk file as *file*, the report writer creates an EDIT file of that name.

If you omit *file*, the report writer closes the current OUT_REPORT file and directs reports to the current OUT file.

CLEAR

clears the new file before anything is written to it. If you omit CLEAR, reports are appended to the existing data in the file. CLEAR is ignored unless the file is a disk file or process.

SPOOL3 *spool-option* [, *spool-option*] ...

specifies one or more level 3 spooling options. The options apply only to spooler files that have not already been opened by previous OUT or OUT_REPORT commands; they have no effect on open files.

The level 3 spooling options (and the defaults SQLCI uses when you open a spooler file) are:

LOC <i>#name</i>	Spooler location. Default is #DEFAULT.
FORM <i>name</i>	Name of form. Default is blanks (no form).
REPORT <i>name</i>	Name of report. Default is your user ID.
COPIES <i>num</i>	Number of copies. Default is 1.
PAGESIZE <i>num</i>	Lines per page in PERUSE. (Make this the same as the PAGE_LENGTH option.) Default is 60.

For more information about level 3 spooling, see the *Spooler Programmer's Guide*.
For more information about PERUSE, see the *Spooler Utilities Reference Manual*.

Considerations—OUT_REPORT

- If there is a current OUT_REPORT file, reports print to that file (and to the current LOG file, if any), but not to the OUT file. If there is no current OUT_REPORT file, reports print to the OUT file.
- SQLCI writes only formatted data to the OUT_REPORT file and ejects a page before and after each report from a SELECT.
- The ENV command displays the name of the current OUT_REPORT file, in addition to other information about the SQLCI session.

Examples—OUT_REPORT

- This example sends reports to a disk file named DRAFT on the current volume and subvolume, clearing the file before writing:

```
>> OUT_REPORT DRAFT CLEAR;
```
- This example sends reports to a spooler location named \$SPLSYS.#S2, naming the report EMPLIST and printing four copies:

```
>> OUT_REPORT $SPLSYS.#S2 SPOOL3 COPIES 4, REPORT EMPLIST;
```

OVERFLOW_CHAR OPTION

OVERFLOW_CHAR is an option of the SQLCI report writer SET STYLE command that specifies the default filler character to print when the value of a numeric report item is too large for its display format.

```
OVERFLOW_CHAR "character"
```

character

is a printable, single-byte character to use as an overflow character.

The default is *.

Consideration—OVERFLOW_CHAR

You can override the default filler character for a specific print item by specifying the OC modifier in the display format for the item. For more information, see [AS Clause](#) on page A-60.

Example—OVERFLOW_CHAR

If you enter this command option, a value that is too large for a 10-byte display field prints as "++++++":

```
>> SET STYLE OVERFLOW_CHAR "+";
```

OWNER FILE ATTRIBUTE

OWNER is a file attribute that identifies the owner of the file. OWNER applies to tables, indexes, protection views, collations, catalogs, programs, and Guardian files.

```
OWNER group-num, user-num
```

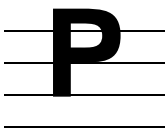
group-num, user-num

specifies the user ID of the user who is to be given ownership of the object;
group-num is an integer that is a group number; *user-num* is an integer that is a user number.

The table default is the user ID of the creating process.

The index default is the user ID of the table's owner.

For more information about file ownership, see [Security](#) on page S-11.



PAGE_COUNT Option

PAGE COUNT is an option of the SQLCI report writer SET LAYOUT command that specifies the maximum number of pages for a printed report.

```
PAGE_COUNT { number }  
           { ALL }
```

number

is an integer in the range 1 through 32,767 that specifies the number of pages to print.

ALL

specifies printing the entire report.

The default is ALL.

Consideration—PAGE_COUNT

After the report writer prints the maximum number of pages, SQLCI terminates the SELECT command that retrieved the information for the report.

Example—PAGE_COUNT

This example limits reports to 40 pages:

```
>> SET LAYOUT PAGE_COUNT 40;
```

PAGE FOOTING Command

PAGE FOOTING is an SQLCI report writer command that specifies text for the bottom of each report page.

```
[PAGE] FOOTING print-item[,print-item...] [ CENTER ] ;
```

print-item

specifies an item to print in the page footing. The form for *print-item* is the same as in the DETAIL command, except that it cannot include the HEADING, NOHEAD, or NAME clause. For more information, see [DETAIL Command](#) on page D-47.

If you specify a column for *print-item*, SQL uses the value of the column in the last detail line on the page.

CENTER

centers each line of the page footing between the left and right margins. If you omit **CENTER**, the page footing starts at the left margin.

Considerations—PAGE FOOTING

- On each page of a report, a blank line separates the page footing from the body of the page. On the last page, the page footing prints below the report footing.
- Only one **PAGE FOOTING** command is in effect at a time. When you enter a **PAGE FOOTING** command, it replaces the previous one.
- The output of the print list you specify in a **PAGE FOOTING** command is a logical line, although (depending on margin settings, device widths, and use of the **SKIP** clause) it might print on more than one physical line. A logical line is limited to 4072 bytes, including the field widths of all print items and the number of spaces between items.

Example—PAGE FOOTING

This example defines a page footing with a page number and current date:

```
S> FOOTING "Page", PAGE_NUMBER AS I2, TAB 45,
+> "Date ", CURRENT_TIMESTAMP AS DATE *;
```

The footing looks like this:

Page 5

Date 11/23/94

PAGE_LENGTH Option

PAGE_LENGTH is an option of the SQLCI report writer **SET LAYOUT** command that specifies the number of lines per page of the report.

<pre>PAGE_LENGTH { ALL } { number }</pre>

ALL

prints the report without page breaks unless you specify a **PAGE** clause in a print list.

The default for reports displayed on a terminal is **ALL**.

The default for other reports is 60.

number

is an integer in the range 1 through 32,767 that specifies the number of lines per report page. *number* must be large enough to print at least one detail line (or a total or subtotal line, if specified) plus any page title and page footing.

Considerations—PAGE_LENGTH

- Each report page begins with the page title (if defined) and ends with a page footing (if defined). The space left for the body of the report (detail lines, headings, subtotals, totals, and so forth) is the page length minus the space used for the page title and page footing and the blank lines that separate the page title and footing from the other lines.
- If you direct a report to a line printer or process, SQL sends a form feed (ASCII 0C or control-L) before each new page. If you direct the report to a disk file, SQL does not send a form feed.

Example—PAGE_LENGTH

This example sets the page length to 66 lines:

```
>> SET LAYOUT PAGE_LENGTH 66;
```

PAGE_NUMBER Function

PAGE_NUMBER is an SQLCI report writer function that returns the page number of the current report page. You can use PAGE_NUMBER in the BREAK FOOTING, BREAK TITLE, DETAIL, PAGE FOOTING, PAGE TITLE, REPORT FOOTING, and REPORT TITLE report writer commands.

PAGE_NUMBER

Considerations—PAGE_NUMBER

- Normally, report pages are numbered from 1, but you can modify the sequence of page numbers with the PAGE clause in a print list or in the DETAIL command. If you do so, PAGE_NUMBER returns the modified number. For more information, see [DETAIL Command](#) on page D-47.
- The default display format for the page number is I11.

Example—PAGE_NUMBER

This example prints a page number on the title line of a report:

```
S> PAGE TITLE TAB 40, "Monthly Report -", PAGE_NUMBER AS I2;
```

The output looks like this:

Monthly Report - 6

PAGE TITLE Command

PAGE TITLE is an SQLCI report writer command that specifies text for the top of each report page.

```
[ PAGE ] TITLE print-item [ ,print-item ] ... [ CENTER ] ;
```

print-item

specifies an item to print in the page title. The form for *print-item* is the same as in the DETAIL command, except that it cannot include the HEADING, NOHEAD, or NAME clause. For more information, see [DETAIL Command](#) on page D-47.

If you specify a column for *print-item*, SQL uses the value of the column in the first detail line on the page. If a detail line is continued from the previous page, SQL uses the value from the SELECT output row that contains the detail line data.

CENTER

centers each line of the page title between the left and right margins. If you omit CENTER, the page title starts at the left margin.

Considerations—PAGE TITLE

- A blank line separates the page title from the body of the report. On the first page, the page title appears above the report title.
- Only one PAGE TITLE command is in effect at a time. When you enter a PAGE TITLE command, it replaces the previous one.
- The output of a PAGE TITLE command is a logical line, although (depending on margin settings, device widths, and use of the SKIP clause) it might print on more than one physical line. A logical line is limited to 4072 bytes, including the field widths of all print items and the number of spaces between items.

Example—PAGE TITLE

This example defines a title for each page of a report:

```
S> TITLE "----- Accounts of:  ", SALESREP, " -----" CENTER;
```

The title looks like this:

```
----- Accounts of:    220 -----
```

Parallel Index Loading

If you create an index on a base table that already contains data, SQL automatically loads the index file with data from the base table. If the index is partitioned, you can specify the `PARALLEL EXECUTION` option on the `CREATE INDEX` statement to direct SQL to load partitions of the index in parallel. A similar clause on the `LOAD` command serves the same function there.

When you execute `CREATE INDEX` or `LOAD` with the `PARALLEL EXECUTION ON` option and the table being indexed or loaded is not empty, SQL starts a record generator (RECGEN) process for each partition of the table and a sort process (SORTPROG) for each partition of the index. Record generator processes read the base table. Sort processes sort the rows and write them to the index.

The default location for the record generator program file is `$SYSTEM.SYSnn.RECGEN`. You can specify a different location with the `=_SQL_RECGEN_node DEFINE`.

Parallel processing uses more processor cycles and disk processes at the same time than serial (nonparallel) processing, and thus might temporarily monopolize system resources.

For best performance, the disk processes for the volumes used should be distributed evenly across all processors.

Specifying certain attributes of the `=_SORT_DEFAULTS DEFINE` can cause problems with multiple sort processes. For more information, see the *FastSort Manual*.

Default Configuration for Parallel Index Loading

You can use the `PARALLEL EXECUTION` clause to specify a configuration file that describes attributes of record generator and sort processes. If you specify the `PARALLEL EXECUTION` clause but do not specify a configuration file, SQL uses these defaults:

- **Priority**—Record generators and sort processes use the same execution priority as the process that creates the index.
- **processor**—If the partition is local, the record generator or sort process runs in the same processor that runs the primary disk process for the partition's disk. If that processor is not available or the partition is remote, the processor is chosen arbitrarily and then in a sequential, circular fashion. (Note that more than one record generator or sort process might run in the same processor.)
- **Scratch file**—By default, the sort process determines a volume for the scratch file. Record generators do not have scratch files.
- **Number of records (sort scratch file size)**—SQL estimates the number of records that each sort process reads as three times the estimated total number of records in the base table divided by the number of partitions in the index:

3 * Number of Records / Number of Partitions

SQL estimates the number of records in the base table by dividing the file size by the record length. The estimate of the number of records is used by the sort process to calculate the scratch file size. For more information on the scratch file size, see the *FastSort Manual*.

- Swap file—The swap file for a sort process defaults to the same volume as the scratch volume if the scratch volume is local. If the scratch volume is not local, the swap file defaults to \$SYSTEM. The swap file for a record generator process is the volume of the partition being read if the partition is local. If the partition is not local, the swap volume is the default swap volume from the =_DEFAULTS DEFINE.

Note. If no configuration information is present, all record generators that read remote partitions swap to the same volume, and all sort processes that write to remote partitions swap to \$SYSTEM. Multiple processes swapping to the same volume might cause disk space and contention problems on that volume so you might want to specify a configuration for parallel index loading.

Specifying Configuration for Parallel Index Loading

The CONFIG option allows you to specify an EDIT file that contains a description of a default configuration or an explicit configuration for both record generators and sort processes.

The configuration file can contain two types of configuration statements: comments and CREATEINDEX statements. Keywords in the configuration file can be in uppercase, lowercase, or mixed-case letters.

The COMMENT statement includes comments in the file. SQL ignores all lines that begin with the word COMMENT or the characters ==. The syntax is:

```
{ COMMENT comment-text }
{ == comment-text }
```

With the CREATEINDEX option you can specify:

- 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. Thus, CREATEINDEX specifies both default and explicit configuration values for record generators and sort processes. The values you specify override any SQL defaults.

Use default values when you want SQL to choose from a set of values (such as scratch volume names or processor numbers) or to apply the same value to all processes (such as the number of records or execution priority). Use explicit values when you want SQL to use particular values for size limitation or performance reasons.

Default and explicit values are not mutually exclusive. You can set up user defaults to be used in most cases and explicit values for one or two unusual partitions. For example, you can explicitly specify sort scratch volumes but use a default configuration to specify a pool of processors in which to run the sort processes. Or you might specify a default pool of volumes to use as scratch files but specify particular scratch volumes for certain partitions. For more information, see the [Sample Configuration File](#) on page P-11.

```
CREATEINDEX {LOCALONLY}
            {BASETABLE} {DEFAULT [node-name]default-attr}
            {INDEX      } {partition attr[,attr]... }
```

default-attr is:

```
[ CPU ( num [, num ] ... ) ]
[ NOSCRATCHON (scratchvol[, scratchvol]...) ]
[ NUMRECS ( number ) ]
[ PRI ( priority ) ]
[ PROGRAM ( file-name ) ]
[ SCRATCH ( scratchvol [, scratchvol ]...) ]
[ SCRATCHON (scratchvol [, scratchvol]...) ]
[ SWAP ( swapvol [, swapvol ] ... ) ]
```

attr is:

```
[ CPU ( num ) ]
[ NOSCRATCHON (scratchvol[, scratchvol]...) ]
[ NUMRECS ( number ) ]
[ PRI ( priority ) ]
[ PROGRAM ( file-name ) ]
[ SCRATCH ( scratchvol [, scratchvol ]...) ]
[ SCRATCHON (scratchvol [, scratchvol]...) ]
[ SWAP ( swapvol ) ]
```

LOCALONLY

directs SQL to run the parallel load operation and all associated sort processes on the node where the operation was initiated. Use the LOCALONLY option to

preserve software behavior available in NonStop SQL/MP versions prior to version 315.

If you do not specify LOCALONLY, each SORTPROG or RECGEN process runs on the node where the associated partition resides.

If you specify LOCALONLY, it must be the first CREATEINDEX statement in the configuration file.

BASETABLE

indicates attributes apply to processes that read the base table partition.

INDEX

indicates attributes apply to sort processes that write to the index partition.

DEFAULT [*node-name*] *default-attr*

specifies a node name and an attribute value for all partitions on a specific node for which another value is not explicitly specified. If you do not specify *node-name*, SQL applies the DEFAULT statement to the node where the parallel index load is initiated.

partition

specifies the name of the volume (including a node, if desired) that contains the partition to which the attributes apply. For example:

```
$MYVOL
\NWREG.$SALES1
```

The default is the local node.

CPU (*num* [, *num*] ...)

is valid only if INDEX or BASETABLE is specified. processor specifies one or more local or remote processors for the record generator or sort process. You can specify multiple processors only for DEFAULT processors.

NOSCRATCHON (*scratchvol* [, *scratchvol*] ...)

is valid only if INDEX is specified. NOSCRATCHON specifies one or more volumes to be excluded as overflow scratch volumes for the sort process. FastSort also uses this list if you do not specify an initial set of scratch volumes (with the SCRATCH option). The NOSCRATCHON option can be used as a DEFAULT specification or for a specific partition. You cannot specify both NOSCRATCHON and SCRATCHON.

If you specify NOSCRATCHON, SQL excludes \$SYSTEM and TMF audit volumes from the overflow set in addition to the volumes in your list. If SMF is installed on your node, you can specify only physical volumes for NOSCRATCHON.

The *scratchvol* specification can include wild-card characters (* and ?).

NUMRECS (*number*)

is valid only if INDEX is specified and specifies the approximate number of records to be loaded into the index partition. This number is used to calculate the scratch file size, as described in the *FastSort Manual*. If this number is too small, the sort process might fail with sort error 30 and file-system error 45 (File is full). Use this attribute if the index is not partitioned evenly across all volumes.

PRI (*priority*)

is valid only if INDEX or BASETABLE is specified, and specifies the priority at which the record generator or sort process is to run.

PROGRAM (*file-name*)

specifies the name of a local or remote SORTPROG object file if BASETABLE is also specified, or specifies the name of a local or remote RECGEN object file if INDEX is also specified. The associated swap volume must reside on the same node as the object file.

SCRATCH (*scratchvol* [, *scratchvol*]...)

is valid only if INDEX is specified. SCRATCH specifies the name of an initial local or remote scratch volume or volumes where the sort process can sort index records. When you specify a list of scratch volumes in a DEFAULT specification, FastSort assigns one volume to each sort process (each associated with a partition) in a sequential fashion. If there are more partitions than volumes, FastSort reuses the list as needed until all partitions have an initial scratch volume assigned.

The SCRATCH option specifies volumes for use as initial scratch volumes. This is not the same as overflow handling. To specify a set of volumes for overflow, use the SCRATCHON option. To request an overflow pool but exclude specific volumes, use the NOSCRATCHON option.

If SMF is installed on the node you specify, *scratchvol* can be a virtual or physical volume. If you specify a virtual volume, FastSort ignores any volumes specified in SCRATCHON or NOSCRATCHON and uses only the virtual volume for both initial and overflow scratch files.

SCRATCHON (*scratchvol* [, *scratchvol*] ...)

is valid only if INDEX is specified. SCRATCHON specifies one or more volumes to be used as overflow scratch volumes for the sort process—in case one or more initial volumes become full. The sort processes also use this list if you do not specify an initial set of scratch volumes (using the SCRATCH option). The SCRATCHON option can be used as a DEFAULT specification or for a specific partition.

You can specify up to 32 volumes, limited by the line length (a maximum of 132 characters). The *scratchvol* specification can include wild-card characters (* and ?).

You cannot specify both NOSCRATCHON and SCRATCHON. If you do not specify either SCRATCHON or NOSCRATCHON, the sort processes consider using any available volume except \$SYSTEM and TMF audit trail volumes. If SMF is installed on your node, you can specify only physical volumes for SCRATCHON.

```
SWAP ( swapvol [ , swapvol ] ... )
```

is valid only if INDEX or BASETABLE is specified, and specifies the name of the volume (including a node, if desired) on which to place the extended segment swap file. For example:

```
$MYVOL  
\NWREG.$SALES1
```

You can specify multiple swap volume names only for DEFAULT swap volumes.

Consideration—Parallel Index Loading

When the total number of table and index partitions approaches 750, parallel index loading might terminate with one of these:

SQL error 1910, sort start error 10
SQL error 1928, record generator error 10.

If one of these errors occurs, increase the PFS space of the SQLCAT object and license the new copy before reissuing your request.

To increase PFS size, use the BINDER product or specify the PFS run-time option.

You should increase the PFS size only when necessary. When you increase the size, save an unmodified copy of SQLCAT.

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

```

Parameters

Parameters let you provide literals for prepared DML statements or command files when you execute the statements or commands (using EXECUTE or OBEY) rather than when you PREPARE or create them. You can use parameters for literals in DML statements compiled with PREPARE or for literals in SQLCI command files.

Typically, you use parameters instead of literals so that you can PREPARE a statement at one time and execute it later—possibly multiple times—substituting different values for each execution with the USING clause of EXECUTE.

If you use named parameters, you can also use the TACL PARAM command or the SQLCI SET PARAM command to assign values to the parameters before you issue an EXECUTE (for a statement) or OBEY (for a command file). In this case, you can also reuse the parameters in subsequent executions of the statement or command file without resetting them.

This diagram shows how to specify a parameter in a DML statement or SQLCI command file. For information about assigning values to parameters, see [EXECUTE Statement](#) on page E-7 or [SET PARAM Command](#) on page S-35.

```
?[param-name] [ [ INDICATOR ] ?[indicator-param] ]

[ TYPE AS { DATETIME [start-dt TO] end-dt } ]
[ { DATE } ]
[ { TIME } ]
[ { TIMESTAMP } ]
[ { INTERVAL start-dt } ]
[ { [ (start-field-precision) ] } ]
[ { [ TO end-dt ] } ]
```

start-dt and *end-dt* are:

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

Only *end-dt* can include the *precision* option for the FRACTION field.

?[*param-name*]

specifies a parameter and, optionally, a name for the parameter. The name must be an SQL identifier.

[INDICATOR] ?[*indicator-param*]

specifies an indicator parameter to use for inserting null values into the database through the parameter, or for handling null values that might be returned to the parameter in host programs. (An indicator parameter with a value less than 0 indicates a null value; an indicator parameter with a value of 0 indicates a nonnull value.)

An indicator parameter has the same format as the parameter it is associated with.

For information on handling null values in dynamic SQL programs, see the SQL/MP programming manual for your host language.

```

TYPE AS { DATETIME [start-dt TO] end-dt }
        { DATE }
        { TIME }
        { TIMESTAMP }
        { INTERVAL start-dt }
          { [ (start-field-precision) ] }
          { [ TO end-dt ] }

```

tells SQL to expect the value entered for the parameter to be a value of the specified date-time or INTERVAL data type. (You cannot use TYPE AS to specify a character or numeric data type for a parameter. Use the CAST function instead.)

A value for a parameter declared with a TYPE AS clause must have a character data type and must be associated with a single-byte character set or the UNKNOWN character set.

For example, This statement insert the character value in ?BIRTHDAY into a table as a DATE value:

```

SET PARAM ?BIRTHDAY "1989-07-31";
INSERT INTO =EMPLOYEES (BIRTHDATE)
VALUES (?BIRTHDAY TYPE AS DATE);

```

If the name of a DATETIME or INTERVAL parameter occurs more than once in a single SQL statement, each occurrence must include an identical TYPE AS clause.

Considerations—Parameters

- You can use an unnamed parameter only in a prepared DML statement. The unnamed parameter—a question mark (?) by itself—is always a distinct parameter even when it occurs multiple times in a statement. When you execute a statement with unnamed parameters, the position of the parameters indicates which values to use for which parameters.

For example, this statement average the salaries of employees in departments with numbers between 1000 and 2000, and between 5000 and 6000:

```

PREPARE AVGSAL FROM
  "SELECT AVG (SALARY) FROM PERSNL.EMPLOYEE
   WHERE DEPTNUM BETWEEN ? AND ?";

EXECUTE AVGSAL USING 1000, 2000;
EXECUTE AVGSAL USING 5000, 6000;

```

- You can use a named parameter only in a prepared DML statement or in an SQLCI command file. Each occurrence of the same parameter name within a statement or an SQLCI session refers to the same parameter. There is no way to qualify a parameter name.

If you use the same parameter name more than once in a single statement, SQL considers each reference to point to the same parameter and assigns each occurrence the same data type, length, and other attributes as the first occurrence.

Assigning a value to the first occurrence of a parameter in the statement automatically assigns a value to the other occurrences also. For example, assume a statement uses five parameters—two named A, two unnamed, and one named B—ordered:

```
?A, ?, ?B, ?, ?A
```

Executing the statement requires only four values, for example:

```
EXECUTE USING 10, 20, 30, 40;
```

because SQL assigns the first value (10) to each of the parameters named A. SQL ignores any additional values in the EXECUTE USING statement, because four values are sufficient to assign values to the parameters in the statement.

Using the same parameter name more than once in a single statement should be done carefully, because it can lead to loss of data in certain cases. For example, during the execution of an INSERT statement, a parameter is assigned the same data type and attributes as the column into which the parameter's value is first inserted. If SQL truncates the parameter value to fit into the column, other occurrences of the parameter also receive the truncated value, even if the columns for those parameters are large enough to hold the complete value.

- The data type of a parameter is derived from the data type of the target column:
 - If the target column has a numeric data type, SQL handles the parameter as DECIMAL(*n*), where *n* is the number of digits in the parameter value.
 - If the target column has a character data type and the target column has the UNKNOWN character set associated with it, SQL handles the parameter as CHAR(*n*), where *n* is the number of bytes in the parameter value.
 - If the target column has a character data type and the target column has a character set other than UNKNOWN associated with it, SQL handles the data type of the parameter as

```
CHAR(n) CHARACTER SET character-set-name
```

where *character-set-name* is the character set specified in the parameter value and *n* is the number of quoted characters in the parameter value. In this case, the parameter value must be a string literal.

- If you omit the TYPE AS clause from a parameter in a date-time or INTERVAL expression, SQL assigns data types to the parameters according to these rules:
 - The data type is INTERVAL if the parameter name is followed by a range of fields and *start-field-precision* is specified.

- The data type is DATETIME if the parameter name is followed by a range of fields and *start-field-precision* is not specified or if the expression has any of these forms:

parameter-name { + | - } *interval-term*

interval-expression + *parameter-name*

date-time-expression - *parameter-name*

- The data type is NUMERIC if the expression takes either of these forms:

parameter-name { + | - } *scalar-value*

{ + | - } *parameter-name*

Example—Parameters

In this example, you can substitute different values for the ?DEPT1 and ?DEPT2 parameters each time you execute the statement:

```
SELECT AVG (SALARY) FROM PERSONL.EMPLOYEE
      WHERE DEPTNUM BETWEEN ?DEPT1 AND ?DEPT2;
```

Then you can prepare the command with the name AVGSAL and execute it like this:

```
EXECUTE AVGSAL USING ?DEPT1 = 2000, ?DEPT2 = 4000;
```

PARTITION Clause

PARTITION is a clause on the ALTER INDEX, ALTER TABLE, CREATE INDEX, and CREATE TABLE statements that defines secondary partitions for a table or index.

```
PARTITION ( partition [ , partition ] ... )

partition is:

    [ \node. ][ $volume. ][ subvol. ] object

        [ | CATALOG catalog | ]
        [ | PHYSVOL volume-name | ]
        [ | EXTENT { size | ( pri-size [, sec-size ] ) } | ]
        [ | MAXEXTENTS integer | ]
        [ | { FORMAT 1 | FORMAT 2 } | ]

        [ FIRST KEY { value } | ]
                      { ( value [ , value ] ... ) } ]
```

[\node.][\$volume.][subvol.] *object*

is the name of a secondary partition. Each fully expanded partition name must be unique in the network, but must have the same subvolume and simple object name as the index or table being partitioned.

CATALOG *catalog*

specifies the name of a catalog to contain the description of the partition. The catalog must be on the same node as the partition. The default is the current default catalog.

PHYSVOL *volume-name*

specifies a physical volume on which to place the secondary partition if SMF is installed on your node. This option overrides SMF. *volume-name* can be either the name of a physical volume or equivalent DEFINE.

Do not include the node name in your volume name.

This option is available only if you specify a virtual volume for *partition*. *volume-name* must belong to the virtual volume you specify.

EXTENT { *size* | (*pri-size* [, *sec-size*]) }

specifies the EXTENT file attribute for the partition. For more information, see [EXTENT File Attribute](#) on page E-30.

MAXEXTENTS *integer*

specifies the MAXEXTENTS file attribute for the partition. For more information, see [MAXEXTENTS File Attribute](#) on page M-2.

{ FORMAT 1 | FORMAT 2 }

indicates the format of the partition.

The default partition format type for tables and indexes is based on the partition array value of the underlying table. For STANDARD and EXTENDED, the default format is 1. For FORMAT2ENABLED, the default format is 2. For relative and entry-sequenced tables, the default format is always 1 because relative and entry-sequenced tables cannot have a FORMAT2ENABLED partition array.

FIRST KEY { *value* | (*value* [, *value*] ...) }

specifies the first primary key or clustering key value that can be stored in the associated partition. FIRST KEY specifies the lowest value for the partition if the column for the value has an ascending collating sequence; it specifies the highest value for the partition if the column has a descending collating sequence.

You must specify a FIRST KEY clause for partitions of indexes and partitions of tables that have user-defined primary keys or clustering keys. (The clause is shown as optional because it does not apply to relative or entry-sequenced files.)

value is a literal or datetime literal that specifies the first value allowed in the associated partition for a column of the key. For an index partition (but not for a table partition), *value* can also be the keyword NULL, representing a null value. (A null value is considered greater than all other values and equal to other null values.)

For a table, the values in the FIRST KEY clause have a one-to-one correspondence with the columns in the primary key or the clustering key of the table. For an index, the values in the FIRST KEY clause have a one-to-one correspondence with the indexed columns in the order specified on the CREATE INDEX statement (not including the keytag column), and the columns of the primary key or clustering key of the underlying table. Each value must have a data type compatible with the data type of the column it corresponds to.

If you specify fewer FIRST KEY values than there are columns, SQL uses the lowest or highest value for the data type of each remaining column. (The lowest value for an ascending column and the highest value for a descending column.) To find the highest or lowest value for a specific data type, see one of these entries:

[Character Data Types](#)

[DATETIME Data Type](#)

[INTERVAL Data Type](#)

[Numeric Data Types](#)

Consideration—PARTITION

Each partition you specify must follow the rules for partitions described in the entry PARTITIONS.

Example—PARTITION

This example shows a CREATE TABLE statement that uses the PARTITION clause:

```
CREATE TABLE \SYS1.$VOL1.SALES.ODETAIL (
  ORDERNUM      NUMERIC (6) UNSIGNED  NO DEFAULT  NOT NULL,
  PARTNUM       NUMERIC (4) UNSIGNED  NO DEFAULT  NOT NULL,
  UNIT_PRICE    NUMERIC (8,2)         NO DEFAULT  NOT NULL,
  QTY_ORDERED   NUMERIC (5) UNSIGNED  NO DEFAULT  NOT NULL,
  PRIMARY KEY ( ORDERNUM , PARTNUM ) )
  CATALOG \SYS1.$VOL1.SALES
  ORGANIZATION KEY SEQUENCED
  PARTITION (
    \SYS1.$VOL2.SALES.ODETAIL
    CATALOG \SYS1.$VOL1.SALES
    EXTENT (16368,64)
    MAXEXTENTS 944
    FIRST KEY 030000
  ,
    \SYS1.$VOL3.SALES.ODETAIL
    CATALOG \SYS1.$VOL1.SALES
    EXTENT (16368,64)
    MAXEXTENTS 944
    FIRST KEY 040000
  ,
    ... --indicates 20 more
    \SYS5.$VOL1.SALES.ODETAIL --partition
    CATALOG \SYS5.VOL1.SALES  --specifications
    EXTENT (16368,64)
    MAXEXTENTS 944
    FIRST KEY 980000 )
  LOCKLENGTH 6
  EXTENT (16368,64)
  MAXEXTENTS 944
  NOPURGEUNTIL OCT 31 2004, 23:59
  NO AUDIT;
```

Some of the attributes specified apply to the entire table and some only to the primary partition. NOPURGEUNTIL, NO AUDIT, and LOCKLENGTH apply to all partitions. The example specifies NO AUDIT because the attribute can be changed to AUDIT after the table will be loaded. The EXTENT and MAXEXTENTS attributes apply only to the primary partition, for which \SYS1.\$VOL1 and a first key of 000000 are assumed.

Partitions

A partition is the portion of a table or index that resides on a single disk volume. Each table or index consists of at least one partition.

An “unpartitioned” table or index is a table or index that consists of exactly one partition. A “partitioned” table or index is a table or index that consists of more than one partition.

A “primary partition” is the first partition in a partitioned table or index. Other partitions are called “secondary partitions.” If the order is ascending, the primary partition contains the lowest set of key values in the table or index; if the order is descending, the primary partition contains the highest set of key values.

A partition name, like a table or index name, is a Guardian name. Each fully expanded partition name must be unique within the network. If a table or index consists of more than one partition, the subvolume and file name portions of the name of each partition in the table or index must be identical. The combination of disk volume and node name will be different for each partition, reflecting the fact that the different partitions reside on different disk volumes.

You can create a partitioned table with CREATE TABLE. For a table with key-sequenced file organization, you can use the PARTONLY MOVE clause of ALTER TABLE to break the table into partitions or to break a partition into additional partitions. For a table with entry-sequenced or relative file organization, you can add a partition to the end of the table with the ADD PARTITION clause of ALTER TABLE. To create a partitioned index, use CREATE INDEX or the PARTONLY MOVE clause of ALTER INDEX.

A partition's format is based on the partition array value of its table. If the table's partition array value is STANDARD, its partitions will be Format 1. If it is FORMAT2ENABLED, its partitions can be of either format, although they will default to Format 2. A Format 1 partition's size is limited to two gigabytes. Format 2 partitions can be up to one terabyte or the limit of a single disk volume, whichever is less.

You can configure tables and indexes that are Format 2 enabled, which lets them have Format 1 or Format 2 partitions, with CREATE TABLE and CREATE INDEX. You can also add new partitions of either format to existing tables and indexes with ALTER TABLE and ALTER INDEX. Use the MOVE clause of ALTER TABLE or ALTER INDEX to move data between existing partitions and newly created partitions of either format.

These rules apply to partitions:

- The FIRST KEY value of a new partition cannot duplicate the FIRST KEY value of another partition of the table.
- You cannot create a partition in a nonaudited volume, even if the table that includes the partition is not audited.
- You cannot partition key-sequenced tables that use SYSKEY as the primary key.

- New partitions must comply with the limits on the number and size of partitions. For more information, see [Limits](#) on page L-6.
- You can partition tables of any file organization but you cannot partition a key-sequenced table that has a system-defined primary key (as opposed to a user-defined primary key) unless it also has a clustering key.
- For relative and entry-sequenced tables, SQL determines the set of rows in a partition, depending on the size of the partitions and the size of the rows. For key-sequenced tables and for indexes, you specify the set of rows in each partition with the FIRST KEY clause of the PARTITION clause.
- To take advantage of Format 2 enabling, your system must be version 350 or later.
- Only key-sequenced tables can be Format 2 enabled and can have partitions that are either Format 1 or 2. Relative and entry-sequenced tables can only be Format 1 enabled and must have Format 1 partitions.

PARTNS Table

The PARTNS table is a catalog table that describes the partitions of a table. [Table P-1](#) describes the contents of the PARTNS table.

Table P-1. The PARTNS Table

Column Name	Data Type	Description
1 FILENAME*	CHAR (34)	Name of this partition
2 PRIMARYPARTITION *	CHAR (1)	Y if primary partition N if secondary partition
3 PARTITIONNAME*	CHAR (34)	Name of another partition of the table
4 CATALOGNAME	CHAR (25)	Name of catalog in which partition is registered
5 FIRSTKEY	VARCHAR (3000)	Starting values for each column in primary key

* Indicates primary key

The PARTNS table was created in Version 1. There have been no subsequent changes.

A partitioned table has a primary partition (the first partition of the table) and one or more secondary partitions. Each partition has the same subvolume and object name, but a different node or volume name than every other partition of the table (because each partition resides on a different node or volume than every other partition of the table).

Each partition must be registered in a catalog on the node on which that partition resides. For each partition registered in a catalog (the FILENAME column of the PARTNS table), the PARTNS table for that catalog contains one entry for each partition

of the table (the PARTITIONNAME column of the PARTNS table). You can register all partitions of a table on a node in a single catalog if you want, or put them in separate catalogs.

Values in FIRSTKEY are in ASCII format, separated by commas. For example, a key composed of a character and an integer column might have a FIRSTKEY value such as: "A",1234.

Guardian names in the PARTNS table are fully qualified and use uppercase characters.

PERUSE Command

PERUSE is an SQLCI command that invokes the Guardian PERUSE program. Use PERUSE to examine and change the attributes of your spooled jobs in addition to monitoring such jobs while they are in the spooler system. For a complete description of PERUSE, see the *Guardian Utilities Reference Manual*.

```
PERUSE [ /run-options/ ] [supervisor] ;
```

run-options

are one or more standard *run-options*, separated by commas (as described in the *TACL Reference Manual*).

supervisor

is the name of the spooler supervisor that PERUSE communicates with. If supervisor is omitted, PERUSE uses \$SPLS.

Example—PERUSE

This example shows how to start PERUSE from SQLCI:

```
>>PERUSE;
PERUSE - T9101D20 - (01JUN93)      SYSTEM  \SYS
Copyright Tandem Computers Incorporated 1978,1982,1983,1984,
1985,1986,1987,1988,1989,1990,1991
SPOOLER SUPERVISOR IS \SYS.$SPLS
```

JOB	BATCH	STATE	PAGES	COPIES	PRI	HOLD	LOCATION	REPORT
36		READY	2	1	4		#DEFAULT	SALES
533		OPEN		1	4	B	#DEFAULT	BUDGET
1074		READY	1	1	4	A	#DEFAULT	MARCH

Plans

A plan (also called an *execution plan* or a *query execution plan*) is an execution method for a single compiled SQL statement. A plan captures both the semantics and execution characteristics of the statement. Compiled programs typically include many plans. Each plan might be operable or inoperable, optimal or not optimal, and valid or invalid.

An operable plan is a plan that will give correct results for a given set of database tables. An inoperable plan is one whose execution would cause an error, an incorrect query result, or a corruption of the database.

An optimal plan is an operable plan that is also the most efficient plan for processing the statement against a given set of database tables. Not all operable plans are optimal plans. An optimal plan for one set of tables might be operable, but not optimal, for a different, but similar, set of tables.

An invalid plan is a plan considered invalid by SQL because changes made after the plan was compiled might have made the plan inoperable or not optimal. A plan is invalid, for example, if an object referenced in the plan was redefined after the plan was last compiled.

A plan can also be invalid for a specific program startup—but not generally invalid—if the startup-time value of a DEFINE referenced in the plan is different from the value of that DEFINE at the time the plan was compiled.

An invalid plan can also be an operable plan if the set of tables for which the plan was compiled is similar to the set of tables associated with the plan at execution time. For information about the differences permitted between such tables, see [Similarity Checks](#) on page S-54.

An altered plan is an invalid but operable plan that the SQL compiler has updated to use a different set of tables without actually recompiling the plan itself.

For more information about plans, see the *SQL/MP Query Guide*.

POSITION Function

The POSITION function searches for a given substring in a character string. If the substring is found, SQL returns the character position of the substring within the string.

```
POSITION (substring IN character-string [, occurrence])
```

where *substring* and *character-string* are:

```
{ string-literal           }
{ column-name             }
{ param-name              }
{ host-var-name           }
{ UPSHIFT function         }
{ character-expression    }
```

and *occurrence* is:

```
{ numeric-literal         }
{ column-name             }
{ param-name              }
{ host-var-name           }
{ expression              }
```

substring

specifies the string substring to search for in *character-string*.

character-string

specifies the source string.

occurrence

specifies which occurrence of the substring to look for. *occurrence* must have an unsigned numeric data type with a scale of zero. The value of *occurrence* must be greater than zero; otherwise, SQL returns an error. If you omit *occurrence*, SQL searches for the first occurrence of the substring.

Considerations—POSITION Function

- The result is returned as a two-byte signed integer with a scale of zero.
- If the substring is not found in *character-string*, SQL returns 0.
- If the value of *occurrence* is greater than the number of occurrences of *substring* in the string, SQL returns 0.
- If the length of the character string is zero and the length of *substring* is greater than zero, SQL returns 0. If the length of *substring* is zero, SQL returns 1.

- If the length of *substring* is greater than the length of the character string, SQL returns 0.
- If *character-string*, *substring*, or *occurrence* is a null value, SQL returns a null value.
- The collating sequences of *substring* and *character-string* must be the same or comparable, or SQL returns an error. The character sets of *substring* and *character-string* must also be identical.
- To ignore case in the search, use the UPSHIFT function or a collation.

Examples—POSITION Function

- This example returns the value 8:

```
POSITION("John" IN "Robert John Smith")
```

- This example returns the value 12, which is the starting position of the second occurrence of "Hello":

```
POSITION("Hello" IN "Hello, and Hello", 2)
```

- This query returns all records in table EMPLOYEE that contain the substring "Smith" in the EMPNAME column:

```
SELECT * FROM EMPLOYEE WHERE POSITION("Smith" IN EMPNAME) > 0
```

- This query returns all records in table EMPLOYEE that contain the substring "SMITH," regardless of whether the substring is in uppercase or lowercase characters:

```
SELECT * FROM EMPLOYEE
WHERE POSITION ("SMITH" IN UPSHIFT(EMPNAME)) > 0
```

Predicates

A predicate is a statement involving a comparison that evaluates to a value of true, false, or unknown (null). Use predicates within search conditions to specify criteria for choosing rows from tables or views. SQL includes this predicates:

[BETWEEN Predicate](#)

[Comparison Predicate](#) (=, <>, <, >, <=, >=)

[EXISTS Predicate](#)

[IN Predicate](#)

[LIKE Predicate](#)

[NULL Predicate](#)

[Quantified Predicate](#) (ALL, ANY, SOME)

For more information about a specific predicate, see the entry for that predicate.

PREPARE Statement

PREPARE is a dynamic SQL statement and an SQLCI command that compiles an SQL statement for later execution with EXECUTE.

In host programs, PREPARE also returns information to the SQLSA that you can use to declare an SQLDA for DESCRIBE and EXECUTE statements. (For information, see [INCLUDE SQLDA Directive](#) on page I-4 or the SQL/MP programming manual for your host language.)

In SQLCI, you can use PREPARE to check syntax even if you don't intend to execute the statement in the session. PREPARE also returns statistics about the compilation time.

```
PREPARE { stmt-name } FROM { "stmt" | 'stmt' }
        { :stmt-name-var }      { :stmt-var }
```

```
{ stmt-name }
{ :stmt-name-var }
```

specifies a name to be used for the prepared statement. If you specify the name of an existing prepared statement, the new statement overwrites the previous one.

stmt-name is an SQL identifier that is the name. You can use this form to specify the name in programs or in SQLCI. In SQLCI, the name must be unique among other statement and report item names in the SQLCI session.

:stmt-name-var is a host variable that contains an SQL identifier that is the name. You can use this form to specify the name from programs. The variable must be of a type compatible with SQL type CHAR or VARCHAR.

```
FROM { "stmt" | 'stmt' }
      { :stmt-var }
```

specifies the statement to prepare.

```
{ "stmt" | 'stmt' }
```

(for SQLCI only) is a DCL, DDL, DML, or DSL statement enclosed in single or double quotation marks.

:stmt-var

(for programs only) is a host variable of a character data type that contains an SQL statement. The statement:

- Can use parameters as literals if it is a DML statement (see [Parameters](#) on page P-11)

- Cannot refer to host variables
- Cannot use an INTO clause if it is a SELECT
- Cannot be CLOSE, DECLARE CURSOR, DESCRIBE, DESCRIBE INPUT, EXECUTE, EXECUTE IMMEDIATE, FETCH, OPEN, PREPARE, or RELEASE

Considerations—PREPARE

- If a PREPARE statement fails, any subsequent attempt to execute the named statement fails.

Only the process that executes the PREPARE can execute the associated prepared statement. The prepared statement is available for execution until the process (the program or SQLCI session) terminates, executes another PREPARE statement that uses the same statement name (either successfully or unsuccessfully), or (programs only) releases the host variable that contains the statement name with a RELEASE statement.

In programs, the scope of a prepared statement depends partly on the rules of the host language in which the PREPARE executes. For more information, see the SQL/MP programming manual for your host language.

- Unless a CONTROL QUERY BIND NAMES AT EXECUTION directive is in effect when the PREPARE executes, a prepared statement uses defaults and DEFINES in effect at the time it is prepared, not the time it executes. (For information, see [CONTROL QUERY Directive](#) on page C-74 or [Name Resolution](#) on page N-2.)
- You can have up to 20 prepared statements in an SQLCI session. (Programs can have more prepared statements.)

Examples—PREPARE

- This example uses PREPARE to compile an SQL statement stored in :INTEXT, a varying length character variable. The program constructs the SQL statement (not shown), compiles it (naming it OPERATION1), and then executes it.

```
...
EXEC SQL
    PREPARE OPERATION1 FROM :INTEXT;
...
EXEC SQL
    EXECUTE OPERATION1;
...
```

- This SQLCI example prepares a SELECT statement (naming it EMPCOM) and then enters the DISPLAY STATISTICS command to display the preparation statistics:

```
>> PREPARE EMPCOM FROM
+> "SELECT FIRST_NAME, LAST_NAME, DEPTNUM"
+> & "FROM PERSNL.EMPLOYEE WHERE DEPTNUM <> 1500"
+> & "AND SALARY <= (SELECT AVG (SALARY))"
```

```
>> & "FROM PERSONL.EMPLOYEE WHERE DEPTNUM = 1500)";
--- SQL command prepared.
>> DISPLAY STATISTICS;
```

- This SQLCI example prepares an INSERT statement with parameters, then supplies parameter values with the EXECUTE:

```
>> PREPARE EMPIN FROM "INSERT INTO PERSONL.EMPLOYEE"
+> &" VALUES (?, ?, ?, ?, ?, ?)";
---SQL command prepared.
>> EXECUTE EMPIN USING 66, "AMY", "RYAN", 3100, 300, 50500;
      WHERE DEPTNUM = 1500) ";
```

- This SQLCI example uses a string literal within a string literal because the prepared statement includes a string literal itself. (Double quotes delimit the outer string, and two quotation marks represent one quotation mark within the string.)

```
>> PREPARE ADDSUP FROM "INSERT INTO INVENT.SUPPLIER"
+> &" VALUES (?, ?, ?, "BEND", "OREGON", "97709" )";
---SQL command prepared.
>> EXECUTE ADDSUP USING 572, "ULTRA-TECH",
+> &"240 INDUSTRIAL WAY";
--- 1 row(s) inserted.
```

Primary Keys

A primary key is a column or a group of columns whose values uniquely identify the rows in a table and (along with file organization and any collations associated with the file) determine the order in which the rows are stored. Each base table and each index has a primary key.

The primary key of a table stored in a key-sequenced file can be defined by the user (PRIMARY KEY clause of the CREATE TABLE statement), defined by the file system (SYSKEY column), or defined by both the user and the file system (CLUSTERING KEY clause of CREATE TABLE, plus the SYSKEY column).

The primary key of a table stored in a relative or entry-sequenced file is always defined by the file system.

The primary key of an index includes the keytag column, the indexed columns, and—for nonunique indexes—the primary key of the underlying table.

Primary key values affect the order in which rows are stored and retrieved. The length of the primary key is a factor in determining the maximum number of partitions for a table or index. The number of columns in a primary key is a factor in determining the number of columns that can be indexed and the maximum number of indexes possible on a table.

A primary key is sometimes called a *physical primary key*.

For more information, see these entries:

[Clustering Keys](#)

[Index Keys](#)

[Syskeys](#) (system-defined primary keys)

[User-Defined Keys](#) (user-defined primary keys)

Print Item

A print item identifies an item to print in an SQLCI report writer report, optionally accompanied by instructions for formatting the item. A print item can generally be a column identifier for a column in the current SELECT list (a column name, column position number, alias, or detail alias), a literal, an arithmetic expression, or a report clause (for example, SKIP, SPACE, or TAB).

A print list is a set of one or more print items in a report writer command, separated by commas:

```
print-item [ , print-item ] ...
```

For the complete syntax of a print item, see [DETAIL Command](#) on page D-47. See the specific command in which you intend to specify a print item for any special restrictions for that command.

PROGID File Attribute

PROGID is a Guardian file attribute that determines the process accessor ID (PAID) of a process started from the program file. PROGID applies only to program files.

[NO] PROGID

PROGID

sets the PAID of a process started from the file to the Guardian user ID of the file's owner.

NO PROGID

sets the PAID of a process started from the file to the Guardian user ID of the user that starts the process.

The default when a program is created is NO PROGID.

Program Invalidation

An SQL-compiled program that is registered in a catalog is either valid or invalid. A valid program is one that can execute with the current description of the database and that is marked as valid in the file label of the program file and in the PROGRAMS catalog table.

An invalid program requires either explicit or automatic recompilation to execute. An invalid program requires explicit recompilation to revalidate it.

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 the *SQL/MP Programming and Installation Guide*.
- 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.

Certain operations on database objects used by a program or on the file that contains the program cause a program to become invalid. SQL automatically marks programs as invalid in the PROGRAMS catalog table and in the file label when these operations occur. SQL also deletes entries for the program in the USAGES catalog table.

Performing any of these operations on an object used by a program invalidates a registered program:

- ALTER TABLE SIMILARITY CHECK ENABLE (or DISABLE)
- ALTER VIEW SIMILARITY CHECK ENABLE (or DISABLE)
- CLEANUP table, DROP TABLE, or PURGE table
- CLEANUP view, DROP VIEW, or PURGE view
- DROP CONSTRAINT

Preventing Program Invalidation Caused by DDL Operations

Certain DDL operations on an SQL object cause a program that references the object to be invalidated. As a result, 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.

This 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 (including an underlying table of a protection or shorthand view used by the program)
- ALTER TABLE...ADD COLUMN statement (for more information, including restrictions, see the *SQL/MP Installation and Management Guide*.)
- 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
- Changing a collation: dropping and then re-creating the collation, renaming a collation, or changing a DEFINE that points to a collation

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
 - DROP INDEX statement, if the program contains a plan that references the dropped index
-

These operations invalidate specific execution plans within a program even when they do not invalidate the program as a whole. (SQL detects the invalid plans at program execution time by comparing an object's current redefinition timestamp to its compile-time redefinition and then performs similarity checks at that time to determine if the plan requires recompilation.)

Host language compiling, binding, or accelerating a registered SQL program deletes it from the catalog it is registered in if the new version of the program is written to the same object file that held the previous version. You must re-SQL-compile the program to reregister it and make it a valid SQL program.

An unregistered SQL program is technically neither valid nor invalid because SQL does not maintain information about its validity in the file label of the program file or in

the PROGRAMS catalog table. However, the operations listed in the subsections, [Operations That Invalidate a Program](#) on page P-29, and [Preventing Program Invalidation Caused by DDL Operations](#) on page P-29, invalidate specific execution plans within an unregistered program.

For more information about program invalidation, see the *SQL/MP Installation and Management Guide*.

PROGRAMS Table

The PROGRAMS table is a catalog table that describes object programs that have been SQL-compiled. [Table P-2](#) describes the contents of the PROGRAMS table.

Table P-2. The PROGRAMS Table (page 1 of 2)

Column Name	Data Type	Description
1 PROGRAMNAME *	CHAR(34)	Name of file that contains object code
2 GROUPID	SMALLINT UNSIGNED	Group number of file owner's user ID if Guardian program, 0 if OSS program
3 USERID	SMALLINT UNSIGNED	User number of file owner's user ID if Guardian program, 0 if OSS program
4 CREATETIME	LARGEINT SIGNED	Julian timestamp from first SQL-compile
5 SECURITYVECTOR	CHAR(4)	Program's security string if Guardian program, \$\$\$\$ if OSS program
6 RECOMPILETIME	LARGEINT SIGNED	Julian timestamp from last explicit SQL-compile
7 AUTOCOMPILE	CHAR(1)	Y if automatic recompilation allowed N if not
8 VALID	CHAR(1)	Y if program is valid N if not
9 PROGID	CHAR(1)	Y if process accessor ID of running program is to be that of owner of program file N if process accessor ID of running program is to be that of user running program \$ if program in OSS file
10 CLEARONPURGE	CHAR(1)	Y if all data in file is to be physically deleted from disk when file is purged N if data in file is not to be physically deleted from disk when file is purged \$ if OSS program
11 SECURITYMODE	CHAR(1)	S Safeguard security G Guardian security \$ OSS program

Table P-2. The PROGRAMS Table (page 2 of 2)

Column Name	Data Type	Description
12 PROGRAMFORMATVERSION	SMALLINT UNSIGNED	PFV of program (oldest version of SQL that can execute program)
13 PROGRAMCATALOGVERSION	SMALLINT UNSIGNED	PCV of program (oldest version catalog that can register program)
14 FORCE	CHAR(1)	Y if FORCE specified N if not
15 SIMILARITYINFO	CHAR(1)	Y if similarity info stored N if not
16 RECOMPILEMODE	VARCHAR(30)	ALL -RECOMPILEALL ON_DEMAND -RECOMPILEONDEMAND UNKNOWN -pre version 310
17 CHECKMODE	VARCHAR(30)	Value of CHECK clause: INVALID_PROGRAMS (default) INVALID_PLANS INOPERABLE_PLANS
18 REGISTERONLY	CHAR(1)	Y if REGISTERONLY specified N if not
19 OSSFILE	CHAR(1)	Y if OSS file N if not

* Indicates primary key

The columns PROGRAMNAME through CLEARONPURGE (1 through 10) were created in version 1. Columns SECURITYMODE through PROGRAMCATALOGVERSION (11 through 13) were added in version 300. Columns FORCE through REGISTERONLY (14 through 18) were added in version 310. Column OSSFILE (19) was added in version 315.

Guardian names in the PROGRAMS table are fully qualified and use uppercase characters. Names of OSS files are stored as the corresponding ZYQ Guardian names, not OSS pathnames.

Protection View

A protection view is a view defined with the FOR PROTECTION option of the CREATE VIEW statements. The view can be derived from a single table by taking a projection of the columns of the table, a restriction of the rows in the table, or both.

A protection view provides a form of column-level security, because the protection view can be secured independently of the table.

PURGE Command

PURGE is an SQLCI utility that deletes SQL objects (except catalog tables and their indexes), SQL programs in Guardian files, and Enscribe files. For an SQL object, PURGE deletes the file that contains the object, the catalog entries for the object, and objects (but not programs) that depend upon the object.

The local super ID (but not other users) can also use PURGE to delete shadow labels.

You cannot PURGE constraints or catalogs. Use DROP instead.

If PURGE cannot delete objects because of missing objects or bad labels, use CLEANUP instead.

```
PURGE [!] qualified-fileset-list[!] [ [,]option ]... ;
```

option is:

```
{ ALLOWERRORS [ ON | OFF | num ] }
{ [ NO ] LISTALL }
{ SHADOWSONLY }
```

[!] *qualified-fileset-list*[!]

is a qualified fileset list that specifies the items to purge. (See [Qualified Fileset List](#) on page Q-1.)

The leading or trailing exclamation point (!) suppresses the interactive confirmation dialog, which is:

```
DO YOU WISH TO PURGE THE ENTIRE FILESET
<names of groups of files listed vertically>
(Y[ES], N[ONE], S[SELECT], F[ILES]) ?
```

Respond to the prompt with one of these:

YES	Purge the entire set of files
SELECT	Display information about each file (as in BRIEF format from the FILEINFO command) and then prompt again about that file
NONE	Cancel the PURGE command
FILES	Display names of all files that belong to the specified group, then prompt again

If *qualified-fileset-list* includes any partition of a partitioned SQL object or the primary partition of an Enscribe file, SQL purges all partitions of the SQL object or Enscribe file. However, if *qualified-fileset-list* includes the secondary partitions of an Enscribe file, but not the primary partition of the file, SQL purges only the secondary partitions included in the list.

If SMF is installed on your node, *qualified-fileset-list* cannot specify any objects or files on a *\$.ZYS** subvolume.

ALLOWERRORS [ON | OFF | *num*]

specifies the action if errors occur:

- ON Attempt to purge all specified files and objects regardless of how many errors are encountered
- OFF Stop the purge operation after the first error is encountered
- num* Purge all specified files and objects until the number of errors encountered exceeds the value of *num*

If you omit the ALLOWERRORS clause completely, the default is ALLOWERRORS OFF. If you specify ALLOWERRORS but do not specify an option, the default is ALLOWERRORS ON.

If a system error causes a user-defined TMF transaction containing the PURGE command to be rolled back, execution of the PURGE is terminated regardless of the ALLOWERRORS setting.

[NO] LISTALL

specifies whether you want PURGE to display the name of each dropped object in this form:

object-type *\$volume.subvol.name* PURGED

object-type is COLLATION, FILE, INDEX, PROGRAM, SVIEW, PVIEW, or TABLE.

LISTALL is the default. If you specify NO LISTALL, PURGE suppresses the display of confirmations.

SHADOWSONLY

specifies that you want to purge shadow labels within *qualified-fileset-list*, but not other objects or files. If you omit SHADOWSONLY, PURGE does not purge shadow labels. SHADOWSONLY is not allowed within a user-defined TMF transaction.

Considerations—PURGE

- PURGE requires authority to purge objects or files being purged and authority to read and write to all catalogs that describe objects affected by the purge.

To purge an index, you also must be the generalized owner of the underlying table and all partitions of that table must be available.

To purge a table, all partitions, indexes, views, and SQL program files that depend on the table must be available. When you purge any partition of a table, SQL also purges all other partitions of the table, all constraints defined on the table, all

indexes that depend on the table, and all views that depend on the table except dependent shorthand views for which you lack purge authority. SQL invalidates the latter.

To purge a protection view, all partitions of the view, all views and SQL program files that depend on the view, all partitions of the table that the view depends on, and all partitions of all indexes on that table must be available. When you purge a view, SQL also purges all views that depend on the purged view, except for dependent shorthand views for which you do not have purge authority. SQL invalidates the latter.

To purge a collation, you must also purge objects that depend on the collation. If you purge a collation and a table or index that depends on the collation in the same PURGE operation, PURGE deletes the table or index before the collation. An error occurs if you attempt to purge the collation without purging dependent objects.

To purge shadow labels, you must be the local super ID.

- PURGE invalidates any SQL program files that depend on the SQL objects purged.
- You can use PURGE within a user-defined TMF transaction unless the command purges a nonaudited object or a shadow label. If PURGE fails within a user-defined transaction, the entire PURGE operation is undone.

If you use PURGE outside of a user-defined TMF transaction, SQL automatically begins a system-defined transaction for the purge of each audited SQL object in *qualified-fileset-list*. If PURGE fails, only the system-defined transaction in progress at the time of the failure is undone. (SQL does not begin a system-defined transaction for an Enscribe file, even if the file is audited.)

If you press the Break key to interrupt a PURGE operation, SQL reports the last object purged at the time you issued the break request and also completes (but does not report) the purge of the next object to be purged. If a user-defined TMF transaction is not in progress, all changes are committed. If a user-defined transaction is in progress, the transaction is rolled back and all changes are undone.

After pressing the Break key, you can restart the PURGE by reentering the command or by using the FC command. This sequence is permissible:

```
>> PURGE *.*.* FROM CATALOG $VOL1.SUBV1;
>> (press the Break key)
>> PURGE *.*.* FROM CATALOG $VOL1.SUBV1;
```

PURGE operations that involve many partitions, especially remote partitions, can often cause many occurrences of error 73 (The disk file or record is locked) or error 40 (The operation timed out) when the operation attempts to update file labels and catalog entries.

Examples—PURGE

- This example deletes all Enscribe files, SQL programs, and SQL objects (except for catalog table and indexes) on subvolume \$VOL1.PERSNL. The exclamation point suppresses the PURGE confirmation prompt.

```
>> PURGE $VOL1.PERSNL.* ! LISTALL;
```

Depending on the contents of the subvolume, the response might look like this:

```
TABLE $VOL1.PERSNL.DEPT PURGED
TABLE $VOL1.PERSNL.EMPLOYEE PURGED
TABLE $VOL1.PERSNL.JOB PURGED
```

```
3 OBJECT(S) PURGED
```

- This example deletes tables and files on subvolume SALES that have names that begin with the letter “O.” The example assumes that three files meet the criteria and shows the confirmation prompt for the PURGE:

```
>> VOLUME SALES;
>> PURGE O* NO LISTALL;
DO YOU WISH TO PURGE THE ENTIRE FILESET
  $VOL1.SALES.O*
(Y[ES], N[ONE], S[ELECT], F[ILES]) ?S
      CODE EOF   LAST MODIF  OWNER RWEP TYPE REC BLOCK
$VOL1.SALES
ODETAIL      A  12288 30Apr9116:42 255,1NUNU K Ta 14  4096
PURGE ? ( Y/N) Y

ORDERS      A  12288 30Apr9116:41 255,1NUNU K Ta 16  4096
PURGE ? ( Y/N) Y

ORDREP                                255,1NUNU SVi
PURGE ? ( Y/N) N
```

```
2 OBJECT(S) PURGED
```

- This example purges all shadow labels on subvolume SALES but does not purge any other files or objects:

```
>> PURGE $VOL1.SALES.* SHADOWONLY NO LISTALL;
```

PURGEDATA Command

PURGEDATA is an SQLCI utility that clears data from SQL tables and their indexes, from specified partitions of SQL tables without indexes, or from Enscribe files or specified partitions of Enscribe files.

PURGEDATA works on audited and unaudited files but—unlike most other operations on audited files—PURGEDATA cannot be used within a user-defined TMF transaction and cannot be rolled back.

```
PURGEDATA  qualified-fileset-list

[ [ [ , ] ALLOWERRORS [ ON | OFF | num ] ] ]
[ [ [ , ] [ NO ] LISTALL ] ] ;
[ [ [ , ] PARTONLY ] ]
```

qualified-fileset-list

specifies the tables, partitions, and files to clear.

When clearing an entire SQL table, PURGEDATA also clears data from all indexes defined on the table. PURGEDATA does not automatically clear alternate key files of Enscribe files.

If *qualified-fileset-list* includes a primary or secondary partition of an SQL table or the primary partition of an Enscribe file, PURGEDATA clears the entire file or table associated with the partition, including all other partitions, unless you specify the PARTONLY option; if you specify PARTONLY, PURGEDATA clears only the partitions included in *qualified-fileset-list*.

If *qualified-fileset-list* includes a secondary partition of an Enscribe file (but not the primary partition) and you do not specify the PARTONLY option, PURGEDATA does not clear the secondary partition (or any other partitions of that file).

If SMF is installed on your node, *qualified-fileset-list* cannot specify any object or file on a \$*.ZYS*. subvolume.

ALLOWERRORS [ON | OFF | *num*]

specifies what happens if errors occur:

- ON Attempt to clear all specified tables, partitions, and files, regardless how many errors are encountered.
- OFF Stop the operation after a single error.
- num* Clear all specified tables, partitions, and files until the number of errors exceeds *num*.

If you omit the ALLOWERRORS clause completely, the default is ALLOWERRORS OFF. If you specify ALLOWERRORS but do not specify an option, the default is ALLOWERRORS ON.

[NO] LISTALL

specifies whether to list the name of each cleared object.

LISTALL is the default. If you specify NO LISTALL, PURGEDATA suppresses the display.

PARTONLY

specifies that data should be cleared from individual partitions included in *qualified-fileset-list*.

An individual partition to be cleared cannot be part of an SQL table with dependent indexes. In addition, if the partition belongs to an SQL table with a relative or entry-sequenced file organization, it must be the last partition in the file. (Neither of these restrictions applies to Enscribe files.)

Considerations—PURGEDATA

- To use PURGEDATA, you must have write authority for the affected tables and files. For SQL tables or partitions, you must also have authority to read and write to the affected catalogs.
- PURGEDATA cannot clear data from SQL program files, views, collations, or catalog tables.

You cannot use PURGEDATA to clear data directly from an SQL index. (PURGEDATA automatically clears the appropriate indexes when you ask it to clear an SQL table.)

PURGEDATA cannot clear data from an individual partition of an SQL table with dependent indexes. It also cannot clear data from any individual partition but the last one in an SQL table with a relative or entry-sequenced file organization.

- You cannot use PURGEDATA within a user-defined TMF transaction.

SQL does not start a transaction for the entire PURGEDATA operation, but does start a transaction for the portion of the operation that involves changes to file labels and catalogs. As a result, you cannot roll back a PURGEDATA operation, but the consistency of the file labels and catalog entries is protected by TMF.

You can use TMF to recover an audited table or file cleared by PURGEDATA if you have recent online dumps of the table or file and you know the time the table was cleared. Use a TMFCOM RECOVER FILES command with TIME set to a value just before the PURGEDATA operation.

- After pressing the Break key, you can restart a PURGEDATA operation by reentering the same command. This sequence is permissible:

```
>> PURGEDATA *.*.* FROM CATALOG $VOL1.SUBV1;
>> (press the Break key)
>> PURGEDATA *.*.* FROM CATALOG $VOL1.SUBV1;
```

You could also use FC to reenter the PURGEDATA command.

PURGEDATA operations that involve many partitions, especially remote partitions, can often cause many occurrences of error 73 (The disk file or record is locked) or

error 40 (The operation timed out) when the operation attempts to update file labels and catalog entries.

If the PURGEDATA operation fails, PURGEDATA leaves the object or file marked corrupt. To clear the corrupt flag, correct whatever problem caused the operation to fail and repeat the PURGEDATA command.

- PURGEDATA does not apply to audited tables that reside on nodes running versions of SQL/MP software earlier than version 300.
- If a file has dependent indexes, you must first drop them, purge the data, alter the table to drop any partitions, and re-create the indexes.
- 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 accessible to other users.
- In some cases, you might need to reopen the table after the PURGEDATA operation. For example, if there is an Embedded SQL program that has a cursor defined as SELECT query with the BROWSE access option, while fetching rows from this cursor, if data from the table (which this cursor refers) is purged using the PURGEDATA operation, then the FETCH operation might fail with error 8204.

Examples—PURGEDATA

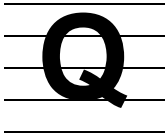
- This example clears all tables and files other than catalog tables or SQL program files on subvolume \$VOL1.PERSNL and lists the names of the cleared files and tables:

```
>> PURGEDATA $VOL1.PERSNL.*;
The confirmation message might look like this:
DATA ARE PURGED FROM TABLE $VOL1.PERSNL.DEPT
DATA ARE PURGED FROM TABLE $VOL1.PERSNL.JOB
DATA ARE PURGED FROM TABLE $VOL1.PERSNL.EMPLOYEE
```

```
DATA ARE PURGED FROM 3 OBJECT(S)
```

- This example clears data from the partition \$NY.SALES.ACCTS without clearing other partitions of the table:

```
>> PURGEDATA $NY.SALES.ACCTS, PARTONLY NO LISTALL;
DATA ARE PURGED FROM 1 OBJECTS(S)
```

Qualified Fileset List

A qualified fileset list specifies a set of objects and files for an SQLCI utility operation and optionally includes clauses that restrict the objects and files operated on based on attributes of the objects and files.

```
{ fileset-list [ restrictions ] }  
{  
  ( fileset-list [ restrictions ]  
    [, fileset-list [ restrictions ] ] ... )  
}
```

fileset-list is:

```
{ fileset }  
{ ( fileset [ , fileset ] ... ) }
```

restrictions is:

```
[ FROM CATALOG[S] catalogs ]  
[ WHERE expression ]  
[ EXCLUDE fileset-list ]  
[ START startfile ]
```

expression is:

```
{ ( expression ) }  
{ NOT expression }  
{ expression AND expression }  
{ expression OR expression }  
{ OWNER = user-id }  
{ timestamp-restriction }  
{ FILECODE { < | <= | = | >= | > | <> } integer }  
{ EOF { < | <= | = | >= | > | <> } integer }  
{ file-attribute }
```

timestamp-restriction is:

```
{ MODTIME } { BEFORE } { ddmmmyyyy [hh:mm:ss] }  
{ EXPIRATIONTIME } { AFTER } { mmdd yyyy [hh:mm:ss] }  
{ CREATIONTIME } { < } { hh:mm:ss }  
{ LASTOPENTIME } { > }
```

fileset

is a set of objects and files specified as a Guardian name that optionally includes these wild-card characters in the volume, subvolume, or file ID portions of the name.

- ? Matches any single character.
For example, TBL? matches TBL1 or TBLX but not TBL48.
- * Matches any 0 to 8 characters.
For example, * matches any 0 to 8 character name; *VOL* matches NEWVOL, OLDDVOL1, VOL45, and so forth.

Notice that a single *fileset* used with wild-card characters can represent a fileset that includes many objects and files.

You cannot use a wild-card character in the node portion of a Guardian name that specifies a *fileset*.

You can use a DEFINE to specify a *fileset*, but you cannot use wild-card characters in the name you specify in the DEFINE. As a result, a *fileset* you specify with a DEFINE always consists of a single object or file.

If SMF is installed on your node, *fileset* cannot specify a file or object on a \$*.ZYS*. subvolume.

FROM CATALOG[S] *catalogs*

restricts operations to files in *fileset* that are also in the specified catalogs. For example, this clause excludes Enscribe files in *fileset* because Enscribe files are not described in any SQL catalog.

Specify *catalogs*:

```
{ catalog-name }
```

```
{ ( catalog-name [ , catalog-name ] ... ) }
```

catalog-name cannot include wild-card characters, but can be a DEFINE.

If SMF is installed on your node, *catalog-name* must be either a logical or direct file.

WHERE *expression*

restricts operations to files in *fileset* that meet the criteria specified by *expression*. *expression* can include parentheses, and NOT, AND, and OR operators. Parentheses have the highest precedence, followed by the others in the order shown.

OWNER = *user-id*

restricts operations to files in *fileset* owned by a Guardian user ID specified:

```
{ group-name.user-name
  group-name.*
  group-number,user-number
  group-number,* }
```

For more information about user IDs, see [Security](#) on page S-11.

timestamp-restriction

restricts operations to files in *fileset* that were created, modified, or last opened before or after a specified date, or that have a NOPURGEUNTIL attribute before or after (or greater than or less than) a specified date and time. (A later date is greater than an earlier date; for example, June 3, 1991 is greater than May 13, 1991.)

Specify dates and times:

mmm	JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC in uppercase or lowercase
dd	an integer from 1 through 31
yyyy	a 4-digit integer from 1900 through 2999
hh	an integer from 0 through 23
mm	a 1 or 2-digit integer from 0 or 00 through 59
ss	a 1 or 2-digit integer from 0 or 00 through 59

You can include leading zeros in *dd*, *hh*, *mm*, or *ss*.

You can include blanks between *dd*, *mmm*, and *yyyy* in *ddmmmyyyy*. You must include a blank between *dd* and *yyyy* in *mmmdyyy*.

The default date is the current date. The default time is 0:00:00.

FILECODE { < | <= | = | >= | > | <> } *integer*

restricts operations to files in *fileset* that have a FILECODE less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to a specified integer.

EOF { < | <= | = | >= | > | <> } *integer*

restricts operations to files in *fileset* that have an EOF less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to a specified integer. (An EOF is a relative byte address that is an end-of-file pointer. You can use FILEINFO to display the EOF value of a file.)

file-attribute

restricts operations to files in *fileset* that have one of these characteristics or file attributes:

AUDITED	INDEX	ROLLFORWARDNEEDED
BROKEN	KEYSEQUENCED	SAFEGUARD
COLLATION	LICENSED	SECONDARY PARTITION
CORRUPT	OPEN	SHORTHAND VIEW
CRASHOPEN	PARTITION	SQLPROGRAM
ENSCRIBE	PRIMARY PARTITION	SQL
ENTRYSEQUENCED	PROGID	TABLE
FORMAT1	PROTECTION VIEW	UNSTRUCTURED
FORMAT2	RELATIVE	VIEW

BROKEN and CRASHOPEN are states described under [FILEINFO Command](#) on page F-9.

FORMAT1 selects files or SQL objects that are in the original disk format.

FORMAT2 selects files or objects that are in the new, larger disk format.

SAFEGUARD selects files with SAFEGUARD ON.

SQL selects files that are SQL tables, views, or indexes.

SQLPROGRAM selects files that are SQL object programs.

Other characteristics and attributes are described in separate entries.

EXCLUDE fileset

specifies a set of files to exclude from the *fileset* you specified just before the WHERE clause with the EXCLUDE clause. EXCLUDE is useful for specifying files from all subvolumes except \$SYSTEM.SYSTEM:

```
$SYSTEM.*.* EXCLUDE $SYSTEM.SYSTEM.*.
```

START startfile

designates a starting position within the files in *fileset*.

START is useful when a utility operation is interrupted and you need to start a new utility operation at the point of the interruption.

SQL utilities process files in the order in which you specify them within *fileset-list*. Each *fileset* within *fileset-list* is processed before the next one. Within the set of files specified by a single *fileset* with wild-card characters, utilities process files in alphabetic order by fully qualified Guardian

name. (To determine the processing order to select the appropriate *startfile*, issue a FILENAMES command with the same qualified fileset list used for the utility.)

startfile is a Guardian file name that can optionally include the wild-card character * as the subvolume or file id portion of the name. The file or set of files that *startfile* specifies must be a subset of the *fileset* to which the clause applies.

Examples—Qualified Fileset List

- This example displays information for all objects on subvolume \$VOL1.PERSNL that are described in the PERSNL catalog:

```
>> FILEINFO $VOL1.PERSNL.* FROM CATALOG $VOL1.PERSNL;
```

- This example displays information for all SQL objects in key-sequenced files on volume \$VOL1 that were created before April 5, 1989:

```
>> FILEINFO $VOL1.*.* WHERE KEYSEQUENCED AND SQL
+> AND CREATIONTIME < APR 5 1989;
```

- This example might be useful after an interrupted SECURE operation that changes ownership of a set of SQL objects. The command changes ownership for only those SQL objects on volume \$VOL1 described in the catalog \$VOL1.SALES, beginning with the file SALES.ODETAIL:

```
>> SECURE $VOL1.*.* FROM CATALOG $VOL1.SALES
+>   START $VOL1.SALES.ODETAIL,
+>   OWNER 302,92;
```

- This example purges objects from the volume \$VOL1 that are registered in the catalog that a previous ADD DEFINE or ALTER DEFINE command specified for the =CAT logical DEFINE name:

```
>> PURGE $VOL1.*.* FROM CATALOG =CAT;
```

Quantified Predicate

A quantified predicate compares the value of an expression to all, some, or any of the values in the result of a subquery.

```
expression comparison-operator [ ANY ]
                                [ ALL ] subquery
                                [ SOME ]
```

comparison-operator is one of the following:

```
=      Equal
<>     Not equal
<      Less than
>      Greater than
<=     Less than or equal to
>=     Greater than or equal to
```

Considerations—Quantified Predicate

- QUANTIFIED is a comparison predicate. For a discussion of general rules for comparisons and specific information about comparing character data (including character data associated with collations), numeric data, date-time data, and interval data, see [Comparison Predicate](#) on page C-58.
- The subquery result must be a table of one column. The data type of the first expression must be compatible with the data type of the subquery result column.
- If you specify ALL, the predicate is true if either of this is true:
 - The comparison is true for every value selected by subquery.
 - The subquery selects no values.
- If you specify ANY, the predicate is true if the comparison is true for at least one value selected by *subquery*. The predicate is false if subquery selects no value, or if the comparison is false for every value selected. SOME is a synonym for ANY.

Specifying =ANY is the same as specifying the IN predicate, although <>ANY is not the same as NOT IN.

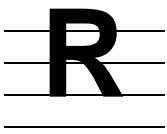
Examples—Quantified Predicate

- This example finds all salaries that are greater than the salaries of all the employees who have a jobcode of 420:

```
SALARY > ALL (SELECT SALARY
               FROM EMPLOYEE WHERE JOBCODE = 420)
```

- This example finds all part numbers that are equal to any part number with more than five units in stock:

```
PARTNUM = ANY (SELECT PARTNUM  
                FROM ODETAIL WHERE QTY_ORDERED > 5)
```

RECLength File Attribute

RECLength is a file attribute that specifies the number of physical bytes of space reserved for each row of a table. RECLength applies only to relative tables.

RECLength <i>length</i>

length

is an integer that specifies the number of bytes to reserve for each record.

RECLength must be at least as great as the total length of all columns in the column list when the table is created and cannot exceed BLOCKSIZE minus 24. (For a table with a block size of 4096, for example, the maximum record length is 4072.)

The default is the length of the table's columns.

Considerations—RECLength

- RECLength reserves space in a relative table so columns can be added later. You cannot add columns to a relative table if the combined length of the new and existing columns exceeds the table's RECLength. You cannot change RECLength after the table has been created.
- Specifying a large RECLength value allows you to add columns later, but wastes disk space until you do. Each row written to the table is allotted the specified space, regardless of the actual length of the current column list.

RELEASE Statement

RELEASE is a dynamic SQL statement that deallocates space in a host program for a dynamic SQL statement that is referenced through a host variable and not declared as a literal in the program.

After a statement is released, references to the statement or to an associated cursor produce errors.

RELEASE : <i>host-identifier</i>

:*host-identifier*

is a host variable of SQL type CHAR or VARCHAR that contains the statement name as declared in the host program. *host-identifier* must conform to the naming conventions of the host language.

For more information, see [Host Variables](#) on page H-6 and the programming manual for your host language.

REPORT FOOTING Command

REPORT FOOTING is an SQLCI report writer command that specifies text for the end of a report.

```
REPORT FOOTING print-item [,print-item]...[ CENTER ] ;
```

print-item

specifies an item to print in the report footing. The form for *print-item* is the same as in the DETAIL command, except that it cannot include the HEADING, NOHEAD, or NAME clause. For more information, see [DETAIL Command](#) on page D-47.

If you specify a column for *print-item*, SQL uses the value of the column in the last detail line in the report.

CENTER

centers each line of the report footing between the left and right margins. If you omit CENTER, the report footing starts at the left margin.

Considerations—REPORT FOOTING

- A blank line separates the report footing from the body of the report. The report footing appears above the page footing on the last page.
- Each REPORT FOOTING replaces the previous REPORT FOOTING. Only one REPORT FOOTING command is in effect at a time. When you enter a REPORT FOOTING command, it replaces the previous one.
- The Print List is limited to 4072 bytes of printed output. The output of a REPORT FOOTING command is a logical line, although (depending on margin settings, device widths, and use of the SKIP clause) it might print on more than one physical line. A logical line is limited to 4072 bytes, including the field widths of all print items and the number of spaces between items.

Example—REPORT FOOTING

This example specifies a report footing that includes both a literal and a column value:

```
S> REPORT FOOTING "End of Summary for Sales Representative",
+> SALESREP CENTER;
```

The footing looks like this:

```
End of Summary for Sales Representative 212
```

REPORT Option

The REPORT option controls generation of EMS messages for an SQL operation started by a statement that includes the option.

REPORT	[ON]
	[OFF]
	[TO <i>collector</i>]

If you specify REPORT without an option, the default is ON.

If you omit the REPORT clause entirely, the default is OFF.

ON

directs event messages for the operation to \$0, the default EMS collector.

OFF

suppresses event messages for the operation.

TO *collector*

directs event messages for the operation to *collector*. *collector* is the name of a primary or alternate EMS collector (or an equivalent DEFINE).

Considerations—REPORT Option

- EMS (Event Management Service) is a collection of processes, tools, and interfaces that provide event-message collation and distribution in the DSM (Distributed Systems Management) environment. A few SQL statements send EMS messages (also called *event messages*) that allow you to monitor the progress of the operation started by the statement. These statements include the REPORT option to allow you to control the generation of such messages and specify the collector to which the messages are sent.

For more information about EMS in general, see the *EMS Manual*. For more information about EMS messages issued by SQL/MP software, see the *SQL/MP Messages Manual*.

- The EDIT file RPTSQL on the subvolume on which NonStop SQL/MP is installed (normally \$SYSTEM.SYSTEM) contains a TACL script that generates EMS reports for SQL operations. You can use it to produce default reports or customize a copy of it to produce variations on the default reports.

For more information, see the comments within the file RPTSQL itself.

Examples—REPORT Option

- This CREATE INDEX example uses the REPORT option to explicitly turn on the sending of EMS messages associated with the operation. Because the REPORT

option does not specify an EMS collector, messages go to \$0, the default EMS collector.

```
CREATE INDEX $DK.REG1.IREL2 ON $DK.APPL.RECORDS(COL1, COL6)
  WITH SHARED ACCESS NAME CREATE_INDEX_IREL2
  REPORT ON
  COMMIT WHEN READY
  ONCOMMITERROR COMMIT BY REQUEST;
```

- The sample report in [Example R-1](#) is produced from EMS messages sent by the operation started with the CREATE INDEX statement in the previous example. The actual report also includes the date and time to the left of the process identification that begins each messages, but the date and time columns are not shown here because of the width of the report.

Example R-1. Report Example (page 1 of 2)

```
\SQ.3,49      TANDEM.SQL.D30      000001
               CREATE_INDEX_IREL2 command started
\SA.3,49      TANDEM.SQL.D30      000002 Target
               \SA.DK.REG1.IREL2 created
\SA.3,49      TANDEM.SQL.D30      000003 All target
               partitions have been created
               Time to create targets: 0 secs
\SA.$X314     TANDEM.SQLAUDIT.D30 000002 ** Audit Fixup
               Status **
               Time since last status: 0 secs
               Distance to audit EOF : ? kbytes
               Records read      : 0 recs
               Records redone    : 0 recs
               Read rate         : 0 bytes/sec
               Redo rate         : 0 bytes/sec
               Progress rate     : ? bytes/sec
\SA.3,49      TANDEM.SQLAUDIT.D30 000001 Audit Fixup
               Initialized
               Time to perform initialization: 6 secs
\SA.3,49      TANDEM.SQL.D30      000004 Data Copy
               Started
\SA.3,49      TANDEM.SQL.D30      000006 Data Copy
               Completed
               Time since last status: 6 secs
\SA.3,49      TANDEM.SQL.D30      000007 Online dump
               allowed for \SA.DK.REG1.IREL2
\SA.3,49      TANDEM.SQLAUDIT.D30 000003 Audit Fixup 1
               started
\SA.3,49      TANDEM.SQLAUDIT.D30 000004 Audit Fixup 1
               completed
               Duration: 2 secs
\SA.3,49      TANDEM.SQLAUDIT.D30 000003 Audit Fixup 2
               started
\SA.3,49      TANDEM.SQLAUDIT.D30 000004 Audit Fixup 2
               completed
               Duration: 2 secs
\SA.3,49      TANDEM.SQL.D30      000017
               CREATE_INDEX_IREL2 command returning to
               caller
               SQLCODE returned: 1618
               Message Text:
               - WARNING from SQL [1618]: The
               CREATE_INDEX_IREL2 statement is ready to
               - commit.
```

Example R-1. Report Example (page 2 of 2)

```

\SA.$X314      TANDEM.SQLAUDIT.D30    000002  ** Audit Fixup
               Status **
               Time since last status: 300 secs
               Distance to audit EOF : ? kbytes
               Records read   : 3781 recs
               Records redone  : 97 recs
               Read rate      : 2370 bytes/sec
               Redo rate       : 28 bytes/sec
               Progress rate   : ? bytes/sec
\SA.$X314      TANDEM.SQLAUDIT.D30    000002  ** Audit Fixup
               Status **
               Time since last status: 300 secs
               Distance to audit EOF : ? kbytes
               Records read   : 4907 recs
               Records redone  : 266 recs
               Read rate      : 677 bytes/sec
               Redo rate       : 51 bytes/sec
               Progress rate   : ? bytes/sec
\SA.3,49       TANDEM.SQL.D30          000017
               CREATE_INDEX_IREL2 command returning to
               caller
               SQLCODE returned: -1622
               Message Text:
               - ERROR from SQL [-1622]: The CREATE_INDEX
               statement specified in the
               - CONTINUE command is not the same as
               the current command in progress.
               - Please enter the correct name.
\SA.3,49       TANDEM.SQL.D30          000009 Commit phase
               has begun
\SA.3,49       TANDEM.SQL.D30          000011 All file locks
               have been obtained
               Time since last event 0 secs
\SA.3,49       TANDEM.SQLAUDIT.D30    000003 Audit Fixup 3
               started
\SA.3,49       TANDEM.SQLAUDIT.D30    000004 Audit Fixup 3
               completed
               Duration: 0 secs
\SA.3,49       TANDEM.SQL.D30          000012 All partition
               labels have been updated,
               Time since last event 2 secs
\SA.$X314      TANDEM.SQLAUDIT.D30    000002  ** Audit Fixup
               Status **
               Time since last status: 105 secs
               Distance to audit EOF : ? kbytes
               Records read   : 4939 recs
               Records redone  : 277 recs
               Read rate      : 26 bytes/sec
               Redo rate       : 9 bytes/sec
               Progress rate   : ? bytes/sec
\SA.3,49       TANDEM.SQLAUDIT.D30    000005 Audit Fixup
               Terminated
\SA.3,49       TANDEM.SQL.D30          000016
               CREATE_INDEX_IREL2 command has completed

```

REPORT TITLE Command

REPORT TITLE is an SQLCI report writer command that specifies text to print at the beginning of the report as the main title for the report.

```
REPORT TITLE print-item [,print-item]...[ CENTER ] ;
```

print-item

specifies an item to print in the report title. The form for *print-item* is the same as for the DETAIL command, except that it cannot include the HEADING, NOHEAD, or NAME clause. For more information, see [DETAIL Command](#) on page D-47.

If you specify a column for *print-item*, SQL uses the column value from the first detail line in the report.

CENTER

centers each line of the report title between the left and right margins. If you omit CENTER, the report title starts immediately after the left margin.

Considerations—REPORT TITLE

- A blank line separates the report title from the body of the report. The report title appears below the page title on the first page.
- Each REPORT TITLE command replaces the previous one. Only one REPORT TITLE command is in effect at a time. When you enter a REPORT TITLE command, it replaces the previous REPORT TITLE command.
- The print list is limited to 4072 bytes of printed output. The output of a REPORT TITLE command is a logical line, although (depending on margin settings, device widths, and use of the SKIP clause) it might print on more than one physical line. A logical line is limited to 4072 bytes, including the field widths of all print items and the number of spaces between items.

Examples—REPORT TITLE

- This example selects data for a report and define list count:


```
>> SET LIST_COUNT 0;
>> SELECT * FROM SALES.ORDERS, PERSONL.EMPLOYEE
+> WHERE SALESREP = EMPNUM AND
+> SALESREP = 226;
```
- This example defines a report title, defines a detail line that omits the SALESREP column, and lists all selected rows. Other elements of the report use the default report format.

```

S> REPORT TITLE "Summary of Orders:", SALESREP,
+> CONCAT (FIRST_NAME STRIP, " ", LAST_NAME);
S> DETAIL COL 1, COL 2, COL 3, COL 5;
S> LIST ALL;
Summary of Orders:  226  HEIDI WEIGL

```

ORDERNUM	ORDER_DATE	DELIV_DATE	CUSTNUM
-----	-----	-----	-----
200490	880319	881101	123
300380	880319	880820	123
600480	880512	881010	3333

Report Writer

The report writer is a component of SQLCI that enables you to produce formatted reports from rows returned by SELECT statements. It includes:

[BREAK FOOTING
Command](#)

[BREAK ON Command](#)

[BREAK TITLE Command](#)

[CANCEL Command](#)

[DETAIL Command](#)

[NAME Command](#)

[OUT_REPORT
COMMAND](#)

[PAGE FOOTING
Command](#)

[PAGE TITLE Command](#)

[REPORT FOOTING
Command](#)

[REPORT TITLE
Command](#)

[RESET LAYOUT
Command](#)

[RESET REPORT
Command](#)

[RESET STYLE Command](#)

[SET LAYOUT Command](#)

[SET STYLE Command](#)

[SUBTOTAL Command](#)

[TOTAL Command](#)

Style and Layout Options

[CENTER_REPORT
Option](#)

[DATE_FORMAT Option](#)

[DECIMAL_POINT Option](#)

[HEADINGS Option](#)

[LEFT_MARGIN Option](#)

[LINE_SPACING Option](#)

[LOGICAL_FOLDING
Option](#)

[NEWLINE_CHAR Option](#)

[NULL_DISPLAY Option](#)

[OVERFLOW_CHAR
OPTION](#)

[PAGE_COUNT Option](#)

[PAGE_LENGTH Option](#)

[RIGHT_MARGIN Option](#)

[SPACE Option](#)

[ROWCOUNT Option](#)

[SUBTOTAL_LABEL
Option](#)

[TIME_FORMAT Option](#)

[UNDERLINE_CHAR
Option](#)

[VARCHAR_WIDTH Option](#)

[WINDOW OPTION](#)

Clauses

AS Clause	IF/THEN/ELSE Clause	SKIP*
AUDIT File Attribute	NEED*	SPACE Option*
CONCAT Clause	PAGE*	TAB*

Functions

COMPUTE_TIMESTAMP Function	LINE_NUMBER Function
CURRENT_TIMESTAMP Function	PAGE_NUMBER Function

For more information, see the entries for specific commands, functions, clauses, or options. Clauses marked with an asterisk (*) are described in [DETAIL Command](#) on page D-47. For more information about using the report writer, see the *SQL/MP Report Writer Guide*.

The uses of these commands, functions, clauses, and options are summarized in the tables that follow.

[Table R-1](#) includes the report writer commands just listed, in addition to other SQLCI commands typically used with report writer commands.

For example, the SQLCI SET SESSION includes such options as MANDATORY_REPORT that can affect report format.

[Table R-2](#) on page R-10, [Table R-3](#) on page R-10, and [Table R-4](#) on page R-11 summarize, respectively, functions, print-list clauses, and style and format options available for report writing.

Table R-1. SQLCI Commands Used to Write Reports (page 1 of 2)

BREAK FOOTING	Defines text for the end of a break group
BREAK ON	Groups lines by item value
BREAK TITLE	Defines text for the start of a break group
CANCEL	Cancels the current SELECT command
BREAK FOOTING	Defines text for the end of a break group
BREAK ON	Groups lines by item value
BREAK TITLE	Defines text for the start of a break group
CANCEL	Cancels the current SELECT command
DETAIL	Defines detail line content and format
EXECUTE	Executes a compiled command
LIST	Displays rows returned by a SELECT command
LOG	Starts or ends logging of session activity to a file
OUT	Specifies or closes the output file

Table R-1. SQLCI Commands Used to Write Reports (page 2 of 2)

OUT_REPORT	Directs the output of a SELECT to a report file or closes the current report file
PREPARE	Compiles a command. Useful for compiling a SELECT command before producing a report
NAME	Defines a name for a column in the select list for use in reports
PAGE FOOTING	Defines text for the bottom of each page
PAGE TITLE	Defines text for the top of each page
REPORT FOOTING	Defines text for the end of the report
REPORT TITLE	Defines text for the start of the report
RESET LAYOUT	Resets layout options to default settings
RESET REPORT	Deletes commands from the current report definition, or deletes columns or alias from stored report formatting commands in the current report definition
RESET SESSION	Resets session options to default settings
RESET STYLE	Resets style options to default settings
SELECT	Retrieves data from tables and views
SET LAYOUT	Sets layout options to new values
SET SESSION	Sets session options to new values
SET STYLE	Sets style options to new values
SHOW LAYOUT	Displays the values of layout options
SHOW REPORT	Displays report formatting commands and the most recent SELECT command
SHOW SESSION	Displays values of session options
SHOW STYLE	Displays values of style options
SUBTOTAL	Defines subtotals
TOTAL	Defines totals

Table R-2. Style and Layout Options for Reports

Option	What the Option Defines	Default
CENTER_REPORT	Is report centered?	OFF
DATE_FORMAT	Format for dates	M2/D2/Y2
DECIMAL_POINT	Symbol for decimal point	Period (.)
HEADINGS	Are headings used?	ON
LEFT_MARGIN	Left margin	0
LINE_SPACING	Number of lines to advance before next report line	1
LOGICAL_FOLDING	Does item of default detail line move to the next line when it does not fit in margins?	ON
NEWLINE_CHAR	Character to indicate a new line in a heading	Slash (/)
NULL_DISPLAY	Character to represent a null value	Question mark (?)
OVERFLOW_CHAR	Filler character printed when a value exceeds the field size	Asterisk (*)
PAGE_COUNT	Maximum pages	ALL
PAGE_LENGTH	Maximum lines per page	ALL for terminal 60 for other devices
RIGHT_MARGIN	Right margin	Device width
SPACE	Spaces between columns	2
ROWCOUNT	Is the row-count line generated?	ON
SUBTOTAL_LABEL	Label to print in a break column with subtotals	Asterisk (*)
TIME_FORMAT	Format for time	HP2:M2:S2
UNDERLINE_CHAR	Character for underlining	Hyphen (-)
VARCHAR_WIDTH	Maximum characters in a varying-length print item	80
WINDOW	Column position displayed at the left edge of output device	TAB 1

Table R-3. Report Writer Clauses (page 1 of 2)

Clause	What the Clause Specifies
AS	Format for a print item
AS DATE/TIME	Format for a printed date or time
CONCAT	Print items without intervening space
IF/THEN/ELSE	Condition for printing items

Table R-3. Report Writer Clauses (page 2 of 2)

Clause	What the Clause Specifies
NEED	Number of lines to keep together on page
PAGE	Advance to the next page and optionally start a new page-number sequence
SKIP	Number of lines before next item
SPACE	Number of blanks between items
TAB	Position of the next item on a line

Table R-4. Report Writer Functions

Function	What the Function Returns
COMPUTE_TIMESTAMP	Timestamp for specified date and time
CURRENT_TIMESTAMP	Timestamp for current date and time
LINE_NUMBER	Current line number within a break group, page, or report
PAGE_NUMBER	Current page number

Reserved Words

[Table R-5](#) lists words that are reserved for NonStop SQL/MP. You cannot use these words as names of constraints, columns (including correlation names), cursors, or statements, but you can use these words in catalog names; in Guardian names that identify tables, indexes, views, partitions, and collations; and in host variable names.

Table R-5. Reserved Words (page 1 of 2)

ALL	CURSOR	IN	NOT	TABLE
AND		INNER	NULL	TO
ANY	DEC	INPUT	NUMERIC	
AS	DECIMAL	INSERT		UNIQUE
ASC	DECLARE	INT	OF	UPDATE
AVG	DELETE	INTEGER	OPEN	
	DESC	INTO	OPTION	VALUES
BEGIN	DISTINCT	IS	OR	VIEW
BETWEEN			ORDER	
BY	ESCAPE	JOIN		WHERE
	EXISTS		RIGHT	WITH
CATALOG		KEY	ROLLBACK	WORK
CHAR	FETCH			
CHARACTER	FOR	LEFT	SAMPLE	

Table R-5. Reserved Words (page 2 of 2)

CHECK	FROM	LIKE	SELECT
CLOSE			SET
COMMIT	GROUP	MAX	SMALLINT
COUNT		MIN	SOME
CURRENT	HAVING		SUM

These words are reserved for the report writer. You should not use these words as user-defined names if your site uses the report writer component of SQLCI:

CURRENT_TIMESTAMP	NOT	SKIP
IF	PAGE	SPACE
LINE_NUMBER	PAGE_NUMBER	TAB
NEED	REPORT	

RESET DEFINE Command

RESET DEFINE is an SQLCI command that restores one or more DEFINE attributes in the working attribute set to their initial settings. (RESET DEFINE is similar to the TACL command ADD DEFINE.)

```
RESET DEFINE { attr [ , attr ] ... } ;
              { *
```

attr

is a DEFINE attribute whose value is to be reset to its initial value. For a list of DEFINE attributes, see [DEFINES](#) on page D-27.

*

resets the DEFINE class to MAP and establishes a working attribute set that consists of the FILE attribute set to its initial value (which is no value).

Considerations—RESET DEFINE

- For a description of the working attribute set, see [SET DEFINE Command](#) on page S-32.
- If an error occurs when you enter a RESET DEFINE command, the command does not change the working attribute set.
- Attributes are reset in the order you specify them.

- When you reset the CLASS attribute, you set the DEFINE class to MAP and establish a working attribute set consisting of the FILE attribute (set to its initial value).
- If you reset an attribute that does not belong to the current class, an error message appears.

Example—RESET DEFINE

This example shows the way RESET DEFINE changes the working attribute set. Initially, the working attribute set contains attributes for CLASS CATALOG. RESET DEFINE resets the SUBVOL attribute value. Because the SUBVOL attribute is required (there is no default value), the attribute is set to no value.

```
>> SHOW DEFINE *;
      CLASS                CATALOG
      SUBVOL                \SYS1.$VOL1.PERSNL
>> RESET DEFINE SUBVOL;
>> SHOW DEFINE *;
      CLASS                CATALOG
      SUBVOL                ??
Current attribute set is incomplete
```

RESET LAYOUT Command

RESET LAYOUT is an SQLCI report writer command that resets layout options to their default settings. Layout options affect the way a report appears on screen or on the printed page.

```
RESET { [ LAYOUT ] option [, option ] ... } ;
      { LAYOUT *
```

The layout options and their default settings are:

CENTER_REPORT	OFF
LEFT_MARGIN	0
LINE_SPACING	1
LOGICAL_FOLDING	ON
PAGE_COUNT	ALL
PAGE_LENGTH	ALL (terminal); 60 (other devices)
RIGHT_MARGIN	Output device width
SPACE	2
WINDOW	TAB 1 (all columns)

* resets all options to default settings.

For more information, see entries for specific options.

Example—RESET LAYOUT

This example resets three report writer layout options to their default values:

```
>> RESET LAYOUT RIGHT_MARGIN, LINE_SPACING, PAGE_LENGTH;
```

RESET PARAM Command

RESET PARAM is an SQLCI command that clears the values of one or more parameters.

```
RESET { [ PARAM ] param-name [ ,param-name ] ... } ;
      { PARAM * }
```

param-name

specifies a parameter to clear.

*

clears the values of all parameters.

For a description of parameters and their use, see [SET PARAM Command](#) on page S-35.

Considerations—RESET PARAM

- The RESET PARAM command clears the current parameter values. You do not have to reset a parameter value that you assign using the EXECUTE command because the assignment applies only to that execution of the command.
- Parameters you set in the command interpreter (with the PARAM command) before starting SQLCI are reset for your SQLCI session but are not changed in the command interpreter.
- After you clear a parameter, the parameter has no value. If you execute a command that refers to the parameter, an error message appears.

Examples—RESET PARAM

- This example clears the values of parameters ?TESTVAL1, and ?TESTVAL2:

```
>> RESET PARAM ?TESTVAL1, ?TESTVAL2
```

- Suppose that you use the TACL PARAM command to set the SAL parameter before starting SQLCI. During your SQLCI session, you set the parameter again. The RESET PARAM command in this example resets ?ENUM and ?STATE to no value. The SAL parameter is also set to no value for the SQLCI session, but when you exit SQLCI, SAL has the value 140,000 for the TACL process. The ?ENUM

parameter has no value after the EXECUTE command terminates because the value is assigned only temporarily.

```

4> PARAM SAL 140000
5> PARAM
DEVICE^TYPE                .2.
SAL                        .140000.
6> SQLCI
SQL Conversational Interface - T9191D20 - (01JUN93)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1987-1994

>> SHOW PARAM *;
?DEVICE^TYPE                2
?SAL                      140000
>> SET PARAM ?ENUM 557;
...
>> SET PARAM ?STATE "CALIFORNIA", ?SAL 45000;
>> SHOW PARAM *;
?ENUM                      557
?STATE                    CALIFORNIA
?SAL                      45000
>> EXECUTE FINDSUP USING ?ENUM = 45;
...
>> SHOW PARAM *;
?ENUM                      557
?STATE                    CALIFORNIA
?SAL                      45000
>> RESET PARAM *;
>> SHOW PARAM *;
>>EXIT

End of SQLCI Session
7> PARAM
SAL                        .140000.

```

RESET PREPARED Command

RESET PREPARED is an SQLCI command that resets prepared commands. Resetting a prepared command is equivalent to deleting it. If you have prepared the maximum of 20 commands, you can reset a command you no longer need to use to prepare another command.

```
RESET PREPARED { command-name [ , command-name ] ... } ;
                { *
```

command-name

is the name you specified when you prepared the command.

*

resets all prepared commands.

Example—RESET PREPARED

- This example resets a prepared command named EMPIN:

```
>> RESET PREPARED EMPIN;
```

RESET REPORT Command

RESET REPORT is an SQLCI report writer command that resets stored report formatting commands (at the select-in-progress prompt, S>) or resets the most recent SELECT command and stored report formatting commands (at the standard SQLCI prompt, >>).

```
RESET REPORT { report-cmd [ , report-cmd ] ... } ;
              { *
              { SELECT
```

report-cmd is:

```
{ BREAK [ ( column [ , column ] ... ) ]
{ BREAK FOOTING [ ( column [ , column ] ... ) ]
{ BREAK TITLE [ ( column [ , column ] ... ) ]
{ DETAIL
{ NAME [ ( alias [ , alias ] ... ) ]
{ [ PAGE ] FOOTING
{ [ PAGE ] TITLE
{ REPORT FOOTING
{ REPORT TITLE
{ SUBTOTAL [ ( column [ , column ] ... ) ]
{ TOTAL [ ( column [ , column ] ... ) ] }
```


report-cmd

is the name of a report-formatting command to reset to its default setting, or (if you specify *column* or *alias* with the command name) with which to delete the specified column name or alias.

You can specify the *report-cmd* option only at the select-in-progress prompt, S>.

The report formatting commands and their default settings are:

BREAK	No breaks
BREAK FOOTING	No break footing
BREAK TITLE	No break title
DETAIL	Default report detail line
NAME	No names
PAGE FOOTING	No page footing
PAGE TITLE	No page title
REPORT FOOTING	No report footing
REPORT TITLE	No report title
SUBTOTAL	No subtotals
TOTAL	No totals

For example, entering “RESET REPORT TITLE, DETAIL;” at the S> prompt deletes the title for the current report and resets the detail line to the default detail line.

The keyword BREAK refers to the BREAK ON command. If you delete a break, you automatically delete the break footings, break titles, and subtotals associated with that break. For example, entering “RESET REPORT BREAK PARTS;” at the S> prompt deletes a break set on the PARTS column and any break footings, break titles and subtotals associated with that break. Other breaks remain in effect.

column

specifies a column to delete from the stored report command. You can specify *column* as a column name, an alias, or COL *number* (which specifies the position of the column in the select list). *column* cannot be a detail alias.

*

resets all report formatting commands to their default settings. If entered at the standard SQLCI prompt (>>), * also resets the most recent SELECT command.

SELECT

deletes the most recent SELECT command from the report definition.

You can specify the SELECT option only at the standard SQLCI prompt, >>.

Consideration—RESET REPORT

You cannot reset a command that defines an alias or detail alias (such as NAME or DETAIL) if other current report commands use the alias or detail alias, so the order in which you reset report commands is significant. For example, if a SUBTOTAL command refers to a detail alias, you must reset the SUBTOTAL command before you reset the DETAIL command. (Note that aliases are reset by RESET REPORT NAME, but detail aliases are reset by RESET REPORT DETAIL.)

Examples—RESET REPORT

- This example uses RESET REPORT at the select-in-progress prompt. SHOW REPORT commands display the stored report formatting commands before and after the RESET REPORT. Note that the * option does not affect the SELECT in effect for the report.

```
S> SHOW REPORT *;
SELECT * FROM INVENT.ODETAIL ORDER BY PARTNUM;
DETAIL PARTNUM HEADING "Part Number", SPACE 5, QTY_ORDERED;
BREAK ON PARTNUM;
SUBTOTAL QTY_ORDERED;
S> RESET REPORT *;
S> SHOW REPORT *;
SELECT * FROM INVENT.ODETAIL ORDER BY PARTNUM;
```

- This example uses RESET REPORT at the standard SQLCI prompt. SHOW REPORT commands display the stored report formatting commands before and after the RESET REPORT. Note that in this example, unlike the previous one, the * option deletes both the report formatting options and the SELECT.

```
>> SHOW REPORT *;
SELECT * FROM INVENT.ODETAIL ORDER BY PARTNUM;
DETAIL PARTNUM HEADING "Part Number", SPACE 5, QTY_ORDERED;
BREAK ON PARTNUM;
SUBTOTAL QTY_ORDERED;
>> RESET REPORT *;
>> SHOW REPORT *;
```

RESET SESSION Command

RESET SESSION is an SQLCI session command that resets SQLCI session options to default settings.

```
RESET [ SESSION ] { option [, option ] ... } ;
                  { *
```

The session options and their default settings are:

AUTOWORK	ON
BREAK_KEY	ON
DISPLAY_ERROR	ALL
ERROR_ABORT	OFF
ERROR_TEXT	DETAIL
LIST_COUNT	ALL
MANDATORY_REPORT	OFF
STATISTICS	OFF
WARNINGS	ON
WRAP	ON

* resets all options to their default settings.

For more information, see [SET SESSION Command](#) on page S-39.

Example—RESET SESSION

This example resets the LIST_COUNT and WARNINGS options:

```
>> RESET LIST_COUNT, WARNINGS;
```

RESET STYLE Command

RESET STYLE is an SQLCI report writer command that resets style options to their default settings. Style options affect the appearance of specific report items, such as underlines, headings, and date and time formats.

```
RESET { [ STYLE ] option [, option ] ... } ;
      { STYLE *
```

The style options and their default settings are:

DATE_FORMAT	M2/D2/Y2
DECIMAL_POINT	.
HEADINGS	ON

NEWLINE_CHAR	/
NULL_DISPLAY	?
OVERFLOW_CHAR	*
ROWCOUNT	ON
SUBTOTAL_LABEL	*
TIME_FORMAT	HP2:M2:S2
UNDERLINE_CHAR	-
VARCHAR_WIDTH	80

* resets all style options to their default settings.

For more information, see the entries for individual options.

Example—RESET STYLE

This example resets the DECIMAL_POINT and NEWLINE_CHAR options:

```
>> RESET DECIMAL_POINT, NEWLINE_CHAR;
```

RESETBROKEN File Attribute

RESETBROKEN is a file attribute that resets the file-label BROKEN flag for a file. RESETBROKEN applies to key-sequenced, relative, and entry-sequenced tables and to indexes. However, you cannot alter the RESETBROKEN attribute for catalog tables.

RESETBROKEN

When you reset the BROKEN flag, it does not invalidate dependent programs.

RIGHT_MARGIN Option

RIGHT_MARGIN is an option of the SQLCI report writer SET LAYOUT command that sets the right margin for the current report and for all subsequent reports until you reset the margin.

RIGHT_MARGIN <i>number</i>

number

is an integer in the range 1 through 255 that indicates the right most byte position in the report line. *number* must be greater than the LEFT_MARGIN layout option.

The default is the width of the output device.

Consideration—RIGHT_MARGIN

Report lines that extend beyond the margin continue on the next line. For default detail lines, the point at which the lines break depends on the LOGICAL_FOLDING option. For other lines, the point at which the lines break depends on the DETAIL command. For other lines, see [LOGICAL_FOLDING Option](#) on page L-53 or [DETAIL Command](#) on page D-47.

Lines might continue on a new line because of the output device width, even if the lines are within the specified right margin. (Such line continuation is controlled by the WRAP option.) Output device widths are:

Terminal	80 bytes
Unstructured disk file	80 bytes
Structured disk file	Record length set at file creation
Edit file	80 bytes (239 if RIGHT_MARGIN is greater than 80)
Process	132 bytes (255 if RIGHT_MARGIN is greater than 132)
Printer	132 bytes (255 if RIGHT_MARGIN is greater than 132)

Example—RIGHT_MARGIN

This example sets the right most byte position of the report to 75:

```
>> SET LAYOUT RIGHT_MARGIN 75;
```

ROLLBACK WORK Statement

ROLLBACK WORK is a transaction control statement that undoes all database modifications made to audited objects during the current TMF transaction, releases all locks on audited objects held by the transaction, and ends the transaction. Without the AUDITONLY clause, ROLLBACK WORK also releases all locks on nonaudited objects locked during the transaction.

ROLLBACK WORK [AUDITONLY]

AUDITONLY

specifies that only locks on audited objects should be released.

If you specify AUDITONLY on a ROLLBACK WORK statement for a transaction that locked nonaudited objects, you must explicitly close cursors or release locks on the nonaudited objects by using CLOSE cursor and UNLOCK TABLE or FREE RESOURCES.

Considerations—ROLLBACK WORK

- TMF transactions begin with BEGIN WORK and end with COMMIT WORK or ROLLBACK WORK. For more information, see [TMF Transactions](#) on page T-6 or [BEGIN WORK Statement](#) on page B-3.
- ROLLBACK WORK returns status information to the SQLCA, so you can use WHENEVER for processing related errors.
- ROLLBACK WORK is equivalent to:
 FREE RESOURCES (an SQL statement)
 ABORTTRANSACTION (a procedure call)

Example—ROLLBACK WORK

This example uses ROLLBACK WORK to terminate a transaction without committing database changes.

The user adds an order for two parts numbered 4130 to the ORDERS and ODETAIL tables, but discovers that there is no such part number while updating the INVENT table to decrement the quantity available.

```
>> BEGIN WORK;
>> INSERT INTO ORDERS VALUES ( 124, 860323, 860330, 75,
7654);
--- 1 row(s) inserted.
>> INSERT INTO ODETAIL VALUES (124, 4130, 250, 2);
---1 row(s) inserted.
>> UPDATE INVENT.PARTLOC SET QTY_ON_HAND = QTY_ON_HAND - 2
+> WHERE PARTNUM = 4130 AND LOC_CODE = "K43";
--- 0 row(s) updated.
>> ROLLBACK WORK;
```

ROLLBACK WORK cancels the inserts that occurred during the transaction and releases the locks held on ORDERS and ODETAIL.

ROWCOUNT Option

ROWCOUNT is an option of the SQLCI SET STYLE report writer command that causes SQLCI to generate or suppress a line that reports the number of rows returned as the result of a SELECT.

ROWCOUNT { ON } { OFF }

ON generates the row-count line.

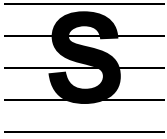
OFF suppresses the row-count line.

The default is ON.

Example—ROWCOUNT

This example shows a simple SELECT with ROWCOUNT ON (the default) and with ROWCOUNT OFF:

```
>> SELECT * FROM MYTABLE;
      aaa    122345
      bbb    333333
      ccc    987653
--- 3 row(s) selected.
>> SET ROWCOUNT OFF;
>> SELECT * FROM MYTABLE;
      aaa    122345
      bbb    333333
      ccc    987653
>>
```

Sample Database

To help users of NonStop SQL/MP become familiar with the product's features, HP includes a sample database and a sample application on the product site update tape (SUT). The sample database demonstrates the use of NonStop SQL/MP in a Pathway transaction processing environment. It includes several host language programs that use embedded SQL statements to access the sample database. Users can also access the sample database with SQLCI commands.

The source code, database creation and load files, and installation instructions for the sample database and application are stored on a file named DOCUMENT on a subvolume named ZTSQMLMSG when NonStop SQL/MP is installed. (The volume name is specified at installation time by the user who installs NonStop SQL/MP. To determine the volume name, check with the group that installs NonStop SQL/MP at your site.) If you set your volume default to the appropriate disk volume, you can use this command to print the installation instructions:

```
35> TGAL /IN ZTSQMLMSG.DOCUMENT, OUT $S.#printer/
```

The sample database is used as the basis for many examples in the NonStop SQL/MP manuals. In the examples, most of the sample database is presented as installed on three subvolumes—PERSNL, SALES, and INVENT—on a single volume. Each subvolume contains a catalog and tables relating to a specific operation in the organization:

PERSNL	Contains the EMPLOYEE, JOB, and DEPT tables, which hold personnel data.
SALES	Contains the CUSTOMER, ORDERS, ODETAIL, and PARTS tables, which are used for order data. Also contains the SUPPKANJ table, which accepts Kanji data for the supplier's name and address.
INVENT	Contains the SUPPLIER, PARTSUPP, PARTLOC, and ERRORS tables, which hold inventory data.

One table in the sample data base—the PARTLOC table—can be installed as a partitioned table. The examples show it as partitioned over three volumes on two different nodes.

For more information about the sample database or the sample application, see the DOCUMENT file. For the host language you use for the illustration "Sample NonStop SQL Database Relations" which shows the relationships between the tables in the sample database, for more information, see the SQL/MP programming manual.

SAVE Command

SAVE is an SQLCI command that saves in a file the values of one or more session attributes in command format. The file can be executed later with an OBEY command.

```

SAVE { ALL
      ENV
      DEFINES
      { [ PARAM ] param-name [, param-name ] ... }
      { PARAM * }
      { [ LAYOUT ] layout-opt [, layout-opt ] ... }
      { LAYOUT * }
      [ SESSION ] { session-opt [, session-opt ] ... }
      { * }
      { [ STYLE ] style-opt [, style-opt ] ... }
      { STYLE * }
      REPORT { report-cmd [, report-cmd ] ... }
      { * }
      COMMAND { "command-string"
                 number
                 -number
              }
      }
      TO file [ ( section-name ) ] [ CLEAR ] ;

```

ALL

saves all the session attributes: environmental parameters, DEFINES, user-defined parameters, session options, layout options, style options, current report formatting commands, and the most recent SELECT command.

ENV

saves the environmental attributes: SYSTEM, VOLUME, CATALOG, OUT, LOG, and, OUT_REPORT.

DEFINES

saves the current =_DEFAULTS DEFINE as an ALTER DEFINE command and saves all other current DEFINES as ADD DEFINE commands.

```

{ [ PARAM ] param-name [, param-name ] ... }
{ PARAM * }

```

saves user-defined parameters as SET PARAM commands; *param-name* is the name of a user-defined parameter. You can specify a list of parameter names to

save specific parameters or specify an asterisk (*) to save the names and values of all current parameters.

```
{ [ LAYOUT ] layout-opt [, layout-opt ] ... }
{ LAYOUT * }
```

saves the layout options you specify; *layout-opt* is a single layout option. An asterisk (*) specifies all layout options. For more information about layout options, see [SET LAYOUT Command](#) on page S-34.

```
[ SESSION ] { session-opt [, session-opt ] ... }
{ * }
```

saves the session options you specify; *session-opt* is a single session option. An asterisk (*) specifies all session options. For more information about session options, see [SET SESSION Command](#) on page S-39.

```
{ [ STYLE ] style-opt [, style-opt ] ... }
{ STYLE * }
```

saves the style options you specify; *style-opt* is a single style option. **STYLE *** specifies all style options. For more information about style options, see [SET STYLE Command](#) on page S-45.

```
REPORT { report-cmd [, report-cmd ] ... }
{ * }
```

saves the report formatting commands you specify and, if specified, the most recent SELECT command. *report-cmd* is one of these commands: BREAK, BREAK FOOTING, BREAK TITLE, DETAIL, NAME, PAGE FOOTING, PAGE TITLE, REPORT FOOTING, REPORT TITLE, SELECT, SUBTOTAL, and TOTAL. **REPORT *** specifies all these commands.

If you enter multiple BREAK TITLE, BREAK FOOTING, NAME, or SUBTOTAL commands, all versions are saved. To save a specific version of a command from the history buffer, enter **SAVE COMMAND** *command-string*.

```
COMMAND { "command-string" }
        { number }
        { -number }
```

saves commands stored in the history buffer.

command-string

is a character string that specifies the most recent command in the history buffer that begins with the string.

number

is a positive integer that refers to the ordinal number of a command in the history buffer.

-number

is a negative integer that indicates the position of a command in the history buffer relative to the current command.

For more information, see [HISTORY Command](#) on page H-5.

TO *file*

specifies a disk file, process file, or terminal name. If the file does not exist, SQLCI creates an EDIT file.

section-name

is the simple name of a section header of the form ?SECTION *section-name*. SQLCI writes the section header in the line preceding the attribute values you are saving.

CLEAR

directs SQLCI to clear the disk or the process file before saving the commands. If you omit this parameter, SQLCI appends the commands to existing text.

You cannot clear individual sections; SQLCI clears the whole file.

The SAVE command is useful for creating an OBEY command file that contains DEFINES you use repeatedly or a report definition you have created using the report options and commands.

Example—SAVE

Suppose that you enter these commands during your SQLCI session:

```
>> VOLUME \SYS1.$VOL1.INVENT;
>> LOG SUBV2.MAYLOG;
>> SYSTEM \SYS1;
>> ADD DEFINE =EMPLOYEE, CLASS MAP,
+> FILE \SYS1.$VOL1.PERSNL.EMPLOYEE;
>> ADD DEFINE =ORDERS, CLASS MAP,
+> FILE \SYS1.$VOL1.SALES.ORDERS;
>> ADD DEFINE =CAT, CLASS CATALOG,
+> SUBVOL \SYS1.$VOL1.PERSNL;
```

You want to set up this session environment and add these DEFINES each time you execute a particular query. To save the commands, enter:

```
>> SAVE ENV TO SETUP1 CLEAR;
>> SAVE DEFINES TO SETUP1;
```

The first command saves the VOLUME, LOG, and SYSTEM commands in the file SETUP1. The next command appends all the ADD DEFINE commands to SETUP1. To set up the environment and DEFINES, enter:

```
>> OBEY SETUP1;
```

Search Conditions

A search condition is a set of predicates (or other search conditions) combined with logical operators (AND, OR, or NOT) that specifies criteria for choosing rows from tables or views.

You can use a search condition in the WHERE clause in a SELECT, DELETE, or UPDATE statement; in the HAVING clause in a SELECT statement; in the ON clause in a SELECT statement that involves a join; in the CHECK clause in the CREATE CONSTRAINT statement; and in the *select-statement* portion of a subquery or of the INSERT or CREATE VIEW statement.

```
[ NOT ] { p } [ { AND } [ NOT ] { p } ] ...
          { (p) } [ { OR } { (p) } ]
```

p is:

```
BETWEEN
Comparison predicate (=, <>, <, >, <=, or >=)
EXISTS
IN
LIKE
NULL
Quantified predicate (ANY, ALL, or SOME)
search-condition
```

NOT

reverses the truth value of the subsequent predicate or search condition.

BETWEEN Comparison predicate, EXISTS, IN, LIKE, NULL, and Quantified predicate

are predicates that specify conditions that must be satisfied for a row to be operated on. For more information, see the entries for specific predicate types.

search-condition

is another search condition.

OR

specifies that the search condition is true if either of the surrounding predicates or search conditions are true.

AND

specifies that the search condition is true only if both the surrounding predicates or search conditions are true.

Considerations—Search Conditions

- SQL evaluates search conditions in this order: first to last; predicates within parentheses; NOT, AND, and OR.
- Within a search condition, a reference to a column refers to the value of that column in the row evaluated by the search condition.
- If a search condition contains a predicate of the form

expression comparison-operator subquery

and the subquery returns no values, the predicate evaluates to null.

If you include a subquery in a search condition, SQL applies the subquery to each row of the table that is the result of the previous clauses, then uses that result to evaluate the search condition in relation to a specific row.

- A statement that contains a search condition operates on a row only if that row satisfies the search condition. For example, in a DELETE statement, any row that satisfies the search condition specified in the WHERE clause is deleted. In a SELECT statement, from each row or group of rows that satisfies the search condition, the columns specified in the select list are returned.

A search condition connected by the OR operator might execute successfully although it includes a predicate that can evaluate to false or null. If any predicate is true, the values of the remaining predicates are irrelevant.

- If a search condition contains a predicate of the form

expression comparison-operator subquery

and the subquery returns no values, the predicate evaluates to null.

For example, this predicate evaluates to null because the subquery returns no value (there is no part number with more than 1500 units in stock):

```
PARTNUM = (SELECT PARTNUM
           FROM ODETAIL
           WHERE QTY_ORDERED > 1500)
```

Examples—Search Conditions

- This example searches for values in rows where the quantity is less than 9, the delivery date is before November 2, 1991 and the order number in the ORDERS table equals the order number in the ODETAIL table:

```
QTY_ORDERED < 9 AND DELIV_DATE <= 911101
AND ORDERS.ORDERNUM = ODETAIL.ORDERNUM
```

- This example searches for values where supplier number in the SUPPLIER table equals supplier number in the PARTSUPP table, and part number is less than 3000 or equal to 7102:

```
SUPPLIER.SUPPNUM = PARTSUPP.SUPPNUM
AND (PARTNUM < 3000 OR PARTNUM = 7102)
```

SECURE Command

SECURE is an SQLCI utility that changes security, ownership, and some file attributes for tables, views, collations, SQL programs in Guardian files, and Enscribe files.

SECURE does not change the security or ownership of catalogs, catalog tables, or SQL programs in OSS files.

```
SECURE qualified-fileset-list [ [,] option ] ... ;
```

option is:

```
{ "rwep"
  ALLOWERRORS [ ON | OFF | num ]
  CLEARONPURGE [ ON | OFF ]
  OWNER { group-name.user-name
        { group-number, user-number }
  }
  PROGID [ ON | OFF ]
  [ NO ] LISTALL
}
```

qualified-fileset-list

is a qualified fileset list that specifies the SQL objects, SQL programs in Guardian files, and Enscribe files for which to change security or ownership. For information, see [Qualified Fileset List](#) on page Q-1.

SECURE changes the security you specify for a table or protection view before it changes the ownership. SECURE changes the security of objects, programs, and files in the order in which you specify them.

To change the security of a partitioned Enscribe file, you must specify the primary partition in *qualified-fileset-list*. SECURE then changes the security for all partitions.

If SMF is installed on your node, *qualified-fileset-list* cannot specify any file or object on a \$*.ZYS*. subvolume.

"rwep"

is a four-character string that specifies the new Guardian security for the objects and files. For more information, see [Security](#) on page S-11.

Make sure the security you specify for an SQL object includes read access for users who have write access.

If you do not specify a security string, the security of files and objects remains unchanged.

`ALLOWERRORS [ON | OFF | num]`

specifies action when errors occur:

ON Resecure all elements of the fileset list regardless of how many errors are encountered.

OFF Stop the operation when an error occurs.

num Resecure files and objects until the number of errors is greater than *num*.

If you omit the `ALLOWERRORS` clause completely, the default is `ALLOWERRORS OFF`. If you specify `ALLOWERRORS` but do not specify an option, the default is `ALLOWERRORS ON`.

If a user-defined transaction that includes a `SECURE` operations is rolled back, the `SECURE` operation is terminated regardless of the setting of `ALLOWERRORS`.

`CLEARONPURGE [ON | OFF]`

specifies whether to erase the contents of an object or file when the object or file is purged from disk.

ON Physically delete the data from the disk by overwriting the file space with binary zeros.

OFF Logically deallocate the disk space but do not physically destroy the data.

`CLEARONPURGE` does not apply to collations.

If you omit the `CLEARONPURGE` option, the file attribute is not changed.

`CLEARONPURGE ON` is the default if you specify `CLEARONPURGE` but omit `ON` and `OFF`.

`OWNER { group-name.user-name }`
`{ group-number, user-number }`

specifies the Guardian user ID of the user who will be given ownership of the object or file. For more information about Guardian user IDs, see [User IDs](#) on page S-12.

If you do not specify the `OWNER` option, the ownership of the object remains unchanged.

PROGID [ON | OFF]

determines the process accessor ID of a program file when the program executes.

ON Set the process accessor ID to the Guardian user ID of the owner of the program.

OFF Set the process accessor ID to the Guardian user ID of the user who runs the process.

If you omit the PROGID option, the file attribute is not changed.

PROGID ON is the default if you specify PROGID but omit ON and OFF.

[NO] LISTALL

specifies whether you want SECURE to display the name of each resecured file or object in this form:

object-type \$*volume.subvol.name* SECURED

object-type is COLLATION, FILE, PROGRAM, PVIEW, SVIEW, or TABLE.

LISTALL is the default. NO LISTALL suppresses the display.

Considerations—SECURE Command

- Authorization and accessibility requirements

To resecure an object or file, you must be the generalized owner of the object or file. You must also have read and write authority to the catalogs for the objects being resecured. To alter the security attributes of a program file, you must also have read authority to the program file.

All partitions, dependent indexes, and dependent protection views for tables being changed must be accessible when SECURE executes.

- Effect on related objects

Altering the security or ownership of an object can affect related objects. For more information, see the ALTER commands for specific types of objects (ALTER TABLE, ALTER VIEW, and so forth).

- Transactions, breaks, and failures

If you use SECURE within a user-defined TMF transaction, all resecuring of audited objects is reversed if SECURE fails during execution.

If you use SECURE outside of a user-defined TMF transaction, SQL automatically begins a system-defined transaction for each SQL object you resecure. In this case, only the resecuring of a single object is undone if SECURE fails.

The SECURE operation is not protected by the TMF subsystem for an Enscribe file.

You can press the Break key to interrupt the SECURE utility. SECURE reports the last object resecured. If a user-defined TMF transaction is not in progress, the changes made to the database before you pressed the Break key are committed and the change in progress at the time you press the key is also committed, although SECURE does not issue a message confirming the last change. If a user-defined transaction is in progress, the transaction is rolled back and all changes are undone.

After pressing the Break key, you can restart the operation by entering the same command again as shown:

```
>> SECURE *.* FROM CATALOG $VOL1.SUBV1 "NUUU";
>> (press the Break key)
>> SECURE *.* FROM CATALOG $VOL1.SUBV1 "NUUU";
```

You can also use the FC command to repeat the same SECURE command.

Note that restarting a partially-completed SECURE operation from the beginning can cause errors if the operation changes ownership. You might no longer have authority to specify ownership for the files whose ownership has already changed. You can avoid this problem by using the START option in *qualified-fileset-list* to restart the operation from the point at which it stopped.

Examples—SECURE Command

- This command resecures all SQL objects from catalog \$VOL1.PERSNL located on subvolume DEPT or JOB so that users on other nodes in the network can read them but only the owner can write to or purge them:

```
SECURE (DEPT.*, JOB.*) FROM CATALOG $VOL1.PERSNL, "NUUU";
```

- This command changes the security and sets the CLEARONPURGE attribute of all tables, views, and programs from catalog CAT on subvolume SV:

```
SECURE SV.* FROM CATALOG CAT, "NUUU" CLEARONPURGE ON;
```

- Each of this SECURE commands uses a different form of a Guardian user ID to change the owner of a table:

```
SECURE $VOL1.PERSNL.EMPLOYEE,, OWNER DEPT3.MGR NO LISTALL;
SECURE $VOL1.PERSNL.EMPLOYEE,, OWNER 9,001 NO LISTALL;
```

SECURE File Attribute

SECURE is a Guardian file attribute that corresponds to the security string that controls Guardian file security for table, index, collation, and protection views.

SECURE "*rwep*"

"*rwep*"

is a four-character string that specifies the Guardian read, write, execute, and purge security for an object or file. For more information, see [Security](#) on page S-11.

Considerations—SECURE File Attribute

- To provide user groups with write access to tables, views, indexes, or catalogs, you must specify read access in addition to write access for those groups.
- A SECURE attribute that permits access to a file from other nodes in the network is necessary, but not sufficient, to allow users to access the file over the network. Network users must also be authorized to access the node on which the file resides in addition to authorized to access that specific file.

Examples—SECURE File Attribute

- This SECURE attribute permits other users on the network to execute the file but permits only the owner to read, write, or purge the file:

```
SECURE "OONO"
```

- This SECURE attribute permits other users on the network to read or update the file but permits only the owner to purge or execute the file:

```
SECURE "NNOO"
```

Security

Authorization to access SQL/MP objects is maintained by the Guardian environment and checked by Nonstop SQL/MP. Each SQL/MP object has associated security values that determine who can read, write to, execute, and purge the object.

SQL programs in Guardian files and other Guardian files used with SQL/MP applications also use the security provided by the Guardian environment. Each Guardian file has an associated set of security values like those for SQL/MP objects.

SQL/MP objects, SQL programs in Guardian files, and other Guardian files can optionally use the Safeguard security management facility, a product that provides security features beyond those of standard Guardian security. The Safeguard

subsystem can secure SQL/MP objects at the volume or subvolume level and can secure all other Guardian files at the volume, subvolume, or file level.

SQL programs in OSS files and other OSS files use OSS security, which differs from Guardian security. A user who runs an SQL program in an OSS file has both an OSS identity (which determines the user's authorization to access OSS files) and a corresponding Guardian identity (which determines the user's authorization to access SQL/MP objects and other Guardian files).

The remainder of this entry describes the general principles of Guardian security as they relate to access to SQL/MP objects, including access from both Guardian and OSS SQL/MP programs. For more information about Guardian, Safeguard, and OSS security, see the *Guardian User's Guide*, the *Safeguard Reference Manual*, and the *OSS User's Guide*.

User IDs

Each user authorized to log on to a node in a NonStop network is identified by a Guardian user ID that consists of a Guardian group number and Guardian user number and that corresponds to a Guardian group name and Guardian user name.

The Guardian user ID is the combination of the group number and user number (not the user number alone) or the combination of the group name and user name (not the user name alone). The user ID is normally represented in one of these forms:

8,55 Group number, comma, user number

DEVEL.JIM Group name, period, user name

A user of a NonStop system must specify a Guardian user ID and an accompanying password to log on to a system through a TACL process. A user who uses the OSS environment of a system can also have a different form of user ID for the OSS environment, but each OSS user ID or alias is associated with a Guardian user ID of the form just described.

Group Manager and Super ID

Each Guardian group includes one special user ID called the group manager that has user number 255 and normally (by convention) has the user name MANAGER; for example:

8,255 Typical group manager ID number

DEVEL.MANAGER Typical group manager ID name

The group manager can act as the owner of any object or file owned by another member of the group. Each node has one special user ID called the super ID that has Guardian group 255 and user number 255. Normally (by convention), both group 255 and user 255 in group 255 are named SUPER; for example:

255,255	Super ID number
SUPER.SUPER	Typical super ID name

The super ID can act as the owner of any object or file on the node. Certain operations can be performed only by a user logged on with the super ID.

Process Access IDs

Each executing process on a system has a process access ID (PAID) that determines the SQL/MP objects and Guardian files the process can access. The process access ID is always a Guardian user ID.

If you work through TACL, the executing TACL process has a process access ID that is the Guardian user ID you supplied at logon. If you work through an OSS shell, the executing shell process has a process access ID that is the Guardian user ID you supplied at logon.

After logon, each process you start normally inherits the processor access ID of the process that starts it—so processes you start from the TACL process, such as SQLCI or host language programs (and any processes you start from those processes), normally inherit the processor access ID that is also the Guardian user ID you supplied at logon. In this way, your initial logon usually determines the SQL/MP objects and Guardian files that you can access.

A process does not inherit the processor access ID of the process that starts it if you execute a program that has the PROGID file attribute set. The PROGID attribute of a program file specifies that a process started from that program file should use the Guardian user ID of the owner of the program file as its process access ID, not the process access ID of the user who starts the process. When this occurs, the Guardian user ID of the owner of the program file determines the SQL/MP objects and Guardian files that the program can access, regardless of the user that executes the program.

The process access ID of the process you are executing (with the [Security Strings](#) on page S-14) determines the objects and files you can access with that process. Therefore, if the SQL documentation says that to perform a certain operation

"you must have authority to ..."

it means that the process access ID of the process you execute must have the authority. A Group List is associated with a process. Each Guardian user can be a member of one or more user groups. The Group List is a list of decimal numbers, specifying the user groups to which the Guardian user belongs. The Group List is always associated with the creator accessor ID (CAID) of a process, even if the process is started from a PROGID object file. The Group List is also used (together with the PAID) to determine the objects and files that you can access with that process.

The owner of an SQL program in a Guardian file can use the ALTER PROGRAM statement or the SECURE command to set the PROGID attribute of the program file. If a program is secured with the Safeguard subsystem, the owner can use the Safeguard

subsystem to set the PROGID attribute. NonStop SQL/MP stores the PROGID attribute of an SQL program in the PROGRAMS table of the catalog in which the program is registered and in the file label of the program itself.

File Ownership

Each SQL/MP object or Guardian file is owned by a single Guardian user ID. When an object or file is created, the owner is the Guardian user ID that corresponds to the process access ID of the process that created the file. You can change the owner of an SQL object with an appropriate ALTER command or with the SECURE command.

A *generalized owner* of an object or file is any user ID that has ownership privileges for the file. On the node where the file is located, the generalized owner always includes the user ID that owns the file, the group manager of the group that includes that user ID, and the super ID. If the owner can purge the file from another node in the network (as specified with the fourth character of the security string described next), the generalized owner also includes the same owner user ID on other nodes, the group manager on other nodes, and the super ID on other nodes.

Security Strings

Each SQL/MP object or Guardian file is associated with a four-character security string, *rwep*, that controls access to that object or file:

r	Read access	(SELECT)
w	Write access	(INSERT, UPDATE, or DELETE)
e	Execute access	(EXECUTE)
p	Purge access	(DROP)

Each character in the security string can have one of these values:

Value	Users Allowed Access
-	Local super ID only
O	Local owner, local group manager, and local super ID
G	Local group member, local owner, and local super ID
A	Any local user
U	Any member of the owner's user class (local or remote users with the same user ID), local or remote group manager, and local or remote super ID
C	Any member of the owner's community (local or remote users with the same group number) and local or remote super ID
N	Any local or remote user

Local refers to a user logged on to the same node. Remote refers to a user logged on to a different node in the same network. For example, the security string "OOOO" specifies that only the local generalized owner of a file can access the file in any way.

In contrast, the security string “NGNU” specifies that any user on the network can read or execute the file (the “r” and “e” characters of the security string) but the generalized owner or a local user with a user ID that has the same Guardian security group as that of the owner can only write to the file (the “w” character of the security string). Only the generalized owner can purge the file (the “p” character of the security string). The generalized owner includes the owner user ID, group manager, and super ID on other nodes in the network.

Authorization Requirements for SQL Statements

To access an object in a NonStop SQL/MP database, an executing process (an SQLCI session or a host program) must have a processor access ID with the appropriate authority based on the security string associated with the object. Different SQL statements have different authorization requirements.

To determine whether a process can update information in a table, for example, NonStop SQL/MP checks read and write access (checks the process access ID against the “rw” characters in the security string for the table). To determine whether a process can change the definition of a table, NonStop SQL/MP also checks read and write access for the catalog that describes the table (checks the process access ID against the “rw,” characters of the security strings for files in the catalog).

Using SQLCOMP to compile an SQL program requires read and purge authority to the program file; read and write authority to the PROGRAMS, USAGES, and TRANSIDS tables of the catalog where the program will be registered; and read and write authority to the USAGES and TRANSIDS catalog tables of any catalog with a description of a table or view used by the program.

Executing an SQL program requires read and execute authority to the program file. Executions that require dynamic recompilation also require read authority to any catalog with a description of a table or view used by the program.

The authorization requirements in SQL/MP statements and SQLCI commands are described with the specific statement or command entries, but [Table S-1](#) summarizes requirements for the major SQL statements.

Table S-1. Authorization Requirements for SQL Statements (page 1 of 3)

Statement	Authority Required
DCL	
LOCK TABLE UNLOCK TABLE	Read authority to the table or view and to underlying tables of the view
DDL*	
ALTER*	Generalized ownership of the object (or for an index, of the underlying table), program file, or catalog being altered; for a program, you must also have read and write authority to the catalogs that describe the program and the objects referenced in the program

Table S-1. Authorization Requirements for SQL Statements (page 2 of 3)

Statement	Authority Required
COMMENT*	Generalized ownership of the table or view (or for an index, of the underlying table) referenced by the comment
CREATE COLLATION*	Read authority for the collation source file or the collation specified in the LIKE clause and its catalog
CREATE CONSTRAINT*	Generalized ownership and read authority for the underlying table
CREATE CATALOG*	Write authority for the SQL.CATALOGS table on the node where the catalog is to reside
CREATE INDEX*	Generalized ownership and read and write authority for the underlying table; also write authority to the USAGES table of catalogs that describe the underlying table
CREATE TABLE*	Read and write authority to the catalogs that describe the table
CREATE VIEW* (shorthand)	Write authority to the USAGES table of catalogs that describe the underlying tables or views
CREATE VIEW* (protection)	Ownership of the underlying table and read and write authority to that table and all associated indexes unless the view security specifies read and write authority for super ID only
DROP CATALOG*	Read and purge authority to the catalog being dropped; read and write authority to SQL.CATALOGS
DROP CONSTRAINT*	Generalized ownership of the underlying table
DROP INDEX*	
DROP COLLATION*	Purge authority to the object being dropped
DROP PROGRAM*	
DROP TABLE*	
DROP VIEW*	
UPDATE STATISTICS*	Generalized ownership of the table for which the statistics are updated
DML	
DELETE INSERT UPDATE	Read and write authority to the table or protection view being deleted or modified; read authority to the tables, protection views, and underlying tables of shorthand views in subqueries of the statement
OPEN FETCH	Read authority to the tables, protection views, and underlying tables of shorthand views referred to in the SELECT statement that defines the cursor; write authority, too, if the cursor is FOR UPDATE

Table S-1. Authorization Requirements for SQL Statements (page 3 of 3)

Statement	Authority Required
SELECT	Read authority to the tables, protection views, and underlying tables of shorthand views referred to in the statement
INVOKE	Read authority to the catalogs that contain the object descriptions

* All DDL statements require authority to read and write to any catalogs affected by the change in addition to any other requirements listed previously.

SELECT Statement

SELECT is a DML statement that retrieves values from tables and views.

```

SELECT [ ALL | DISTINCT ] select-list

    [ INTO :host-variable [ , :host-variable ] ... ]

FROM  table-ref [ , table-ref ] ...

[ WHERE  search-cond ]
[ HAVING  search-cond ]
[ [ FOR ] { BROWSE | STABLE | REPEATABLE } ACCESS ]
[ [ IN ] { SHARE | EXCLUSIVE } MODE ]
[ GROUP BY { colname } [ collate ] ]
[           { colnum } ]
[ ORDER BY { colname } [ASC[ENDING]] [ collate ] ]
[           { colnum } [DESC[ENDING]] ]

[ UNION [ ALL ] select-statement ]

[ FOR UPDATE OF column-name [ , column-name ] ... ]

select-list is:

    { * } [ { * } ]
    { corr.* } [ { corr.* } ]
    { expr } [ { expr } ]

table-ref is:

    { table }
    { view } [ corr ]
    { join-table }

join-table is:

    table-ref [ INNER ] JOIN table-ref ON search-cond
    [ LEFT ]

collate is:

    COLLATE { collation | CHARACTER SET }

```

[ALL | DISTINCT]

specifies whether to retrieve all rows of the intermediate table described by the FROM clause or only rows that are not duplicates (DISTINCT rows). NULL values are considered equal for removing duplicates.

The default is ALL.

select-list

specifies the columns to select from the intermediate table described by the FROM clause:

List Item	Columns of Intermediate Table Retrieved
<i>*</i>	All columns, including SYSKEYs for view
<i>corr.*</i>	All columns in the table or view implicitly or explicitly associated with correlation name <i>corr</i> except the SYSKEY of a table
<i>expr</i>	The columns required to evaluate the expression <i>expr</i>

expr is an SQL expression that is not a subquery. Columns named in *expr* must be from tables or views specified in the FROM clause but can include system-defined primary keys. (Qualify SYSKEY if you want to select the SYSKEY column from more than one table or view: for example, EMPLOYEE.SYSKEY.)

If you refer to a grouped view in the FROM clause, *expr* cannot include an aggregate function on a column of the grouped view. A grouped view is a view defined with a CREATE VIEW AS clause that contains a GROUP BY or HAVING clause that is not in a subquery, contains an aggregate function in the select list, or refers to a grouped view in the FROM clause.

If *expr* is a single column name or a qualified column name, that column of the result table is a named column. All other columns are unnamed columns.

The *** and *corr.** forms of a *select-list* specification are convenient for use in SQLCI but should be avoided in programs. Such specifications make the order of columns in the result table dependent on the order of columns in the current definition of the referenced tables or views. If columns have been added, the retrieved values might not be in the order the program is coded to use.

INTO :*host-variable* [, :*host-variable*] ...

(allowed in programs only) specifies host variables in which to return the result of a query.

You can use the IN clause only for operations that are not union operations and that return no more than one row. (If the query returns more than one row, use a cursor.) For information on using host variables and handling null values that might be returned to host variables, see the SQL/MP programming manual for your host language.

FROM *table-ref* [, *table-ref*] ...

is a list of up to 16 tables, views, and *join-tables* (or equivalent DEFINES), each optionally qualified by a correlation name, that specifies the contents of an intermediate table from which SQL retrieves the columns you specified in *select-list*.

If you specify only one *table-ref*, the intermediate table consists of rows from that table, view, or *join-table*. If you specify more than one *table-ref*, SQL builds the intermediate table by logically concatenating each row of each table, view, or *join-table* in the list with each row of every other table, view, or *join-table* in the list. Within the intermediate table, tables and views are in the order specified in the FROM clause; columns from each table or view are in the order in which they exist in that table or view.

If *table-ref* is a grouped view, the view must be the only *table-ref* in the FROM clause.

table-ref [INNER] JOIN *table-ref* ON *search-cond*
[LEFT]

specifies a join of a table, view, or *join-table* with another table, view, or *join-table*.

INNER joins all rows that satisfy *search condition*.

LEFT joins all rows that satisfy *search condition* plus rows from *table-ref* that do not.

If you specify LEFT JOIN, the *table-ref* on the right side of the keywords LEFT JOIN (which is called the *inner table* of the join) cannot be a *join-table* or a shorthand view whose definition is based on a join operation or on a union of SELECT statements; the *table-ref* on the left cannot be a shorthand view whose definition is based on a LEFT JOIN operation or on a union of SELECT statements.

search-cond is the search condition for the join. Each column in the search condition must be a column that exists in the intermediate table specified by the FROM clause. If the search condition contains an expression list, the expression list must be enclosed in parentheses, as shown:

```
TABLE1 JOIN TABLE2 ON (A,B) > (10,20)
```

For more information, see [Joins](#) on page J-1.

WHERE *search-cond*

specifies a search condition to apply to each row of the FROM clause result table. Each column that you specify in *search-condition* must be a column in the FROM clause result table.

HAVING *search-cond*

specifies a search condition to apply to each row of the result table of the previous clause. The search condition is applied to each group, or (if there is no GROUP BY clause) to all the rows.

In the search condition, you can specify any column as the argument of a function (for example, AVG (SALARY)). A column that is not in a function must be, however, a column in a GROUP BY clause or a column in a table or view specified in an outer query. For more information about outer queries, see [Subqueries](#) on page S-82.

If the FROM clause specifies a grouped view, you cannot specify a HAVING clause.

[FOR] { BROWSE | STABLE | REPEATABLE } ACCESS

specifies the access mode for the SELECT.

The default is FOR STABLE ACCESS, which allows concurrent use of the database, but limits access to a row while the row is processed.

For more information, see [Access Options](#) on page A-1.

[IN] { SHARE | EXCLUSIVE } MODE

specifies that SHARE or EXCLUSIVE locks be used on accessed rows of the table and of the index, if any, through which the accesses occur.

Use SHARE mode when your process reads data but does not modify it. Specifying STABLE access and SHARE mode ensures greater concurrency.

Use EXCLUSIVE mode when your process reads data and then modifies it with DELETE or UPDATE. Requesting EXCLUSIVE locks on the SELECT prevents other processes from acquiring SHARE locks on the accessed rows between the time of the SELECT and the time of the subsequent DELETE or UPDATE. Such locks by other processes would prevent your process from escalating its own SHARE locks to the EXCLUSIVE locks required for a DELETE or UPDATE operation, causing your process to wait or timeout.

Note that a SELECT locks only the accessed row in the table and the corresponding row in the index used as the access path for the SELECT. Another process using an index-only path with STABLE or REPEATABLE access can lock rows in an index on the table that was not used as the access path for the SELECT but that is affected by the DELETE or UPDATE. If this operation occurs, your process waits or times out although you specified EXCLUSIVE. (You can avoid this scenario by using the LOCK TABLE IN EXCLUSIVE MODE statement to lock the table and all its indexes, but this prevents other processes from accessing any portion of the table while the lock is in effect and might not be the best solution to the problem.)

The IN clause is not allowed for BROWSE ACCESS. In SHARE MODE it is ignored for cursor SELECT operations.

If you omit the IN clause, SQL uses SHARE until an attempt is made to modify the data, escalates the lock to EXCLUSIVE.

For more information, see [Locking](#) on page L-48.

GROUP BY { *colname* | *colnum* }

specifies columns of the result table from the preceding FROM or WHERE clause that define a set of groups in which each group consists of rows with identical values in the specified columns.

For example, if you specify AGE, the result table contains one group of rows with AGE equal to 40 and one group with AGE equal to 50. If you specify AGE and JOB, the result table contains one group for each different age and job code pair. When you refer to a grouping column in a search condition or expression, you refer to a single value because each row in the group contains the same value in the grouping column.

For grouping, all null values are considered equal to one another. The result table of a GROUP BY clause can have only one null group.

If the FROM clause specifies a grouped view, you cannot specify a GROUP BY clause.

colname

is the name of a single column from a table in the FROM clause, optionally qualified by a table name, view name, or correlation name: for example, CUSTOMER.CITY.

colnum

is a positive integer that specifies a column by its position in the *select-list*. Use *colnum* to refer to unnamed columns, such as expressions.

In a GROUP BY clause, you cannot use *colnum* to specify a column that is an expression that contains a function if the argument of the function is a column of the FROM clause result table. You can use *colnum* if the argument is a correlated reference to a column from an outer query.

`COLLATE { collation | CHARACTER SET }`

specifies an alternate collating sequence that determines the ordering of rows in a column specified on a GROUP BY or ORDER BY clause, temporarily overriding the effect of any collation associated with the column as part of its table definition.

collation is the name of an existing collation that specifies a collating sequence and uses the same character set as the associated column.

CHARACTER SET specifies a collating sequence based on the binary value of characters in the column.

`ORDER BY { colname } [ASC[ENDING]] [collate]`
`{ colnum } [DESC[ENDING]]`

specifies the order in which to sort the rows of the result table.

colname and *colnum* are as previously described for the GROUP BY clause, with these additional restrictions:

- If you specify DISTINCT, *colname* must be in *select-list*.
- If you specify a GROUP BY or HAVING clause, the ordering column must also be a grouping column.
- If an ORDER BY clause applies to a union of SELECT statements, the ordering column must be explicitly referenced, outside a function or an expression, in *select-list* of the leftmost SELECT statement.

ASCENDING and DESCENDING specify the sort order. If you specify the ORDER BY clause without ASCENDING or DESCENDING, the default is ASCENDING.

Be sure to specify the ORDER BY clause for each column you need ordered. Otherwise, SQL determines the order of the column and does not guarantee a specific or consistent order of rows. ORDER BY can reduce performance, however, so use it only if you require a specific order.

For ordering a result table on a column that can contain null values, a null value is considered equal to other null values but greater than all other nonnull values.

ORDER BY is valid only for the SELECT part of an INSERT statement or for a SELECT statement used in a cursor declaration in a host program.

`UNION [ALL] select-statement`

specifies a set UNION operation between the result table of this SELECT statement and the result table of another SELECT statement. The *select-lists* in the two SELECT statements must have the same number of columns, and columns in corresponding positions within the *select-lists* must have comparable data types (for example, both numeric or both character types).

The result of UNION is a table that contains rows belonging to either of the two tables. If you specify UNION ALL, the table contains all the rows retrieved by each SELECT statement; otherwise, duplicate rows are removed.

The number of columns in the table is the same as the number of columns in each *select-list*. The column names in the table are the same as the corresponding names in the *select-list* of the leftmost SELECT statement. A column resulting from the UNION or expressions or constants has the name EXPR. For the characteristics of data in the individual columns, see [Considerations for UNION](#) on page S-25.

A UNION operation is not allowed with SELECT INTO.

If the UNION of SELECT statements is part of a view definition or a cursor declaration, the view or cursor cannot be updated.

```
FOR UPDATE OF column [ , column ] ...
```

(only for dynamic SQL statements that are not subqueries) associates a list of updateable columns with the statement so that a cursor can be declared for the statement.

Considerations—SELECT

- SELECT requires authority to read all views and tables referred to in the statement, including the underlying tables of all shorthand views referred to in the statement.
- Queries on audited tables or on audited or mixed views must be performed in a TMF transaction unless the SELECT statement specifies BROWSE ACCESS. For more information, see [TMF Transactions](#) on page T-6.
- A view can be considered a select specification saved in a catalog. When a view is referenced in a SELECT statement or a subquery, the select specification that defines the view is combined with the statement or the subquery. The combination can cause the SELECT statement or the subquery to be invalid.

If you receive an error message that indicates a problem but your SELECT statement or subquery appears valid, check the view definition. For example, a view named AVESAL includes column A defined as AVG(X). A SELECT statement that contains MAX(A) in its select list is invalid because the select list actually contains MAX (AVG(X)), and a function cannot have an argument that includes another function.

To determine if a query using functions and GROUP BY clauses is valid, use SQLCI to query the view definition in the TEXT column of the VIEWS catalog table, as shown:

>> SET VARCHAR_WIDTH 225;	Sets wide report line
>> SELECT TEXT FROM \$v.sv.VIEWS	From VIEWS table
+> WHERE VIEWNAME = "view-name";	Fully qualified name in uppercase letters

A grouped view is a view defined with a CREATE VIEW AS clause that contains a GROUP BY or HAVING clause that is not in a subquery, contains an aggregate function in the select list, or refers to a grouped view in the FROM clause.

A shorthand view whose definition is based on a union of SELECT statements or that contains a LEFT JOIN operator, cannot participate in another join operation. A shorthand view whose definition contains any join operation cannot be the inner table of a LEFT JOIN.

- If you include a GROUP BY clause, the columns you refer to, in expressions in the select list, must be either grouping columns or arguments of a function. There must be no more than one row in the result table for each group.

For example, if AGE is not a grouping column, you can refer to AGE only in a function invocation, such as AVG (AGE). If you do not include a GROUP BY clause, but you specify a function in the select list, all rows of the result table form a group. AVG and SUM cause a single value for the table, and COUNT counts all rows. In this case, the select list can contain only functions because there are no grouping columns.

If you specify a GROUP BY clause, a function applies to each row of each group. The result of AVG and SUM is a value for each group. COUNT returns, for each group, the number of rows in that group.

- Columns containing multibyte characters cannot be displayed on all types of output devices.
- You can alter the `=_SORT_DEFAULTS` define to specify a scratch file name, and certain other options to be used by the SORTPROG process. This technique is useful if you are selecting data from a large table. For more information, see [=_SORT_DEFAULTS DEFINE](#) on page Z-4 or the *FastSort Manual*.

Considerations for UNION

- The contributing SELECT statements are called SELECT1 and SELECT2.
- The contributing tables resulting from the SELECT statements are called TABLE1 and TABLE2.
- The table resulting from the UNION operation is called RESULT.

Characteristics of UNION Columns

For columns in the same ordinal position in TABLE1 and TABLE2:

- If both columns contain character strings, the corresponding column in RESULT contains a character string whose length is equal to the greater of the two contributing columns.
- If both columns contain variable-length character strings, RESULT contains a variable-length character string whose length is equal to the greater of the two contributing columns.

- If both columns are of exact numeric data types, RESULT contains an exact numeric value whose precision and scale are equal to the greater of the two contributing columns.
- If both columns are of approximate (floating point) numeric data types, RESULT contains an approximate numeric value whose precision is equal to the greater of the two contributing columns.
- If both columns are of date-time data types, RESULT contains a DATETIME value whose precision is the most significant start field to the least significant end field from the ranges of DATETIME fields in the contributing columns. For example, if the column in TABLE1 shows YEAR TO DAY and the column in TABLE2 shows MONTH TO MINUTE, the precision of the corresponding column in RESULT is YEAR TO MINUTE.
- If both columns are of INTERVAL data type, RESULT contains an INTERVAL value whose range of fields is the most significant start field to the least significant end field from the ranges of INTERVAL fields in the contributing columns. The range of INTERVAL fields in RESULT must not contain both year-month and day-time fields. (The year-month fields are YEAR and MONTH. The day-time fields are DAY, HOUR, MINUTE, SECOND, and FRACTION.)
- If both columns are described with NOT NULL, RESULT does not allow null values; otherwise, RESULT allows null values.

These restrictions apply for shorthand views using UNION:

- The view cannot participate in a join operation.
- A SELECT operation on the view cannot specify a GROUP BY or HAVING clause.
- A SELECT operation on the view cannot specify aggregate functions on any view column.

ORDER BY clause and UNION operator

In a query containing a UNION operator, the ORDER BY clause defines an ordering on the result of the union. A SELECT statement cannot have an individual ORDER BY clause.

You can specify an ORDER BY clause only as the last clause following the final SELECT statement (SELECT2 in this example). The ORDER BY clause in RESULT specifies the ordinal position of the sort column either by using an integer or by using the column name from the select list of SELECT1.

- This example on SELECT illustrates the correct usage of ORDER BY:

```
SELECT A FROM T1 UNION SELECT B FROM T2 ORDER BY A
```

- This example on SELECT is incorrect, however, because the ORDER BY clause does not follow the final SELECT:

```
SELECT A FROM T1 ORDER BY A UNION SELECT B FROM T2
```

- This example on SELECT is also incorrect:

```
SELECT A FROM T1 UNION (SELECT B FROM T2 ORDER BY A)
```

Because the subquery (SELECT B FROM T2...) is processed first, the ORDER BY clause does not follow the final SELECT.

GROUP BY Clause, HAVING Clause, and the UNION Operator

In a query containing a UNION operator, the GROUP BY or HAVING clause is associated with the SELECT statement that it is a part of (unlike the ORDER BY clause, which is associated with the result of a union operation). The groups are visible in the result table of the particular SELECT statement. The GROUP BY and HAVING clauses cannot be used to form groups in the result of a union operation.

UNION ALL and Associativity

The UNION ALL operation is associative, meaning that these two queries return the same result:

```
(SELECT * FROM TABLE1 UNION ALL SELECT * FROM TABLE2)
  UNION ALL SELECT * FROM TABLE3;
SELECT * FROM TABLE1 UNION ALL
  (SELECT * FROM TABLE2 UNION ALL SELECT * FROM TABLE3);
```

If both the UNION ALL and UNION operators are present in the query, however, the result depends on the order of evaluation. A parenthesized union of SELECT statements is evaluated first, from left to right, followed by a left-to-right evaluation of the remaining union of SELECT statements.

Examples—SELECT

- This SQLCI example retrieves information from the EMPLOYEE table for employees with a job code greater than 500, and employees in departments with numbers less than or equal to 3000, displaying the results in ascending order by job code. No locks are held while the query is processed.

```
>> SET LIST_COUNT 3;
>> SELECT JOBCODE, DEPTNUM, FIRST_NAME, LAST_NAME, SALARY
+> FROM PERSNL.EMPLOYEE
+> WHERE JOBCODE > 500 AND DEPTNUM <= 3000
+> ORDER BY JOBCODE
+> BROWSE ACCESS;
```

JOBCODE	DEPTNUM	FIRST_NAME	LAST_NAME	SALARY
600	1500	JIMMY	SCHNEIDER	26000.00
600	1500	JONATHAN	MITCHELL	32000.00
900	1000	SUE	CRAMER	19000.00
S> LIST NEXT 2;				
900	1500	SUSAN	CHAPMAN	17000.00
900	2000	BILL	WINN	32000.00
S> CANCEL;				

- This SQLCI example displays selected rows grouped by job code in ascending order. The *select-list* contains only grouping columns and functions because each group results in one row.

```
>> SELECT JOBCODE, AVG (SALARY)
+>   FROM PERSNL.EMPLOYEE
+>   WHERE JOBCODE > 500 AND DEPTNUM <= 3000
+>   GROUP BY JOBCODE
+>   ORDER BY JOBCODE;
JOBCODE  EXPR
-----
      600                29000.00
      900                27125.00
--- 2 row(s) selected.
```

- This SQLCI example uses the HAVING clause to accomplish the same result as the previous example in an alternative but equally efficient way:

```
>> SELECT JOBCODE, AVG (SALARY)
+>   FROM PERSNL.EMPLOYEE
+>   WHERE DEPTNUM <= 3000
+>   GROUP BY JOBCODE
+>   HAVING JOBCODE > 500
+>   ORDER BY JOBCODE;
```

- This example selects data from more than one table by specifying the table names in the FROM clause and specifying the condition for selecting rows of the result in the WHERE clause. The condition is called a join predicate.

This query joins the EMPLOYEE and JOB tables by combining each row of the EMPLOYEE table with each row of the JOB table; the result is the Cartesian product of the two tables. The join predicate specifies that any row with equal job codes is included in the result table. All other rows are eliminated.

```
>> VOLUME $VOL1.PERSNL;
>> SELECT JOBDESC, FIRST_NAME, LAST_NAME, SALARY
+>   FROM EMPLOYEE, JOB
+>   WHERE EMPLOYEE.JOBCODE = JOB.JOBCODE AND
+>   EMPLOYEE.JOBCODE IN (900, 300, 420);
```

These logical steps that determine the result of the previous query:

1. Join the tables.

EMPLOYEE Table			JOB Table		
EMPNUM	...	JOBCODE	...	SALARY	JOBDESC
1		100		175500	MANAGER
1		100		175500	PROD SUPV
1		100		175500	SECRETARY
568		300		39500	MANAGER
568		300		39500	PROD SUPV

568	300	39500	900	MANAGER
-----	-----	-------	-----	---------

2. Drop rows with unequal job codes.

EMPLOYEE Table		JOB Table		
EMPNUM	JOBCODE	SALARY	JOBCODE	JOBDESC
1	100	175500	100	MANAGER
207	420	33000	420	ENGINEER
568	300	39500	300	SALESREP

3. Drop rows with job codes not 900, 300 or 420.

EMPLOYEE Table		JOB Table		
EMPNUM	JOBCODE	SALARY	JOBCODE	JOBDESC
75	300	32000	300	SALESREP
178	900	28000	900	SECRETARY
207	420	33000	420	ENGINEER
568	300	39500	300	SALESREP

4. Process select list, leaving only four columns.

EMPLOYEE Table		JOB Table	
JOBDESC	FIRST_NAME	LAST_NAME	SALARY
SALESREP	TIM	WALKER	32000
SECRETARY	JOHN	CHOU	28000
ENGINEER	MARK	FOLEY	33000
SALESREP	DESIREE	EVANS	39500

- This SQLCI example selects from three tables and groups the rows by job code and (within job code) by department number. Only job codes 300, 420, and 900 are selected. The minimum and maximum salary for the same job in each department is computed, and the rows are ordered by maximum salary.

```
>> VOLUME $VOL.PERSNL;
>> SELECT E.JOBCODE, E.DEPTNUM, MIN (SALARY), MAX (SALARY)
+> FROM EMPLOYEE E, DEPT D, JOB J
+> WHERE E.DEPTNUM = D.DEPTNUM AND E.JOBCODE = E.JOBCODE
+> AND E.JOBCODE IN (900, 300, 420)
+> GROUP BY E.JOBCODE, E.DEPTNUM
+> ORDER BY 4;
```

- This example presents two ways to select data about orders by customers from California. The price for the total quantity ordered is computed for each order number.

```
>> VOLUME $VOL.SALES;
>> SELECT ORDERNUM, SUM (QTY_ORDERED * PRICE)
+> FROM PARTS P, ODETAIL O
```

```

+> WHERE O.PARTNUM = P.PARTNUM AND ORDERNUM IN
+> (SELECT ORDERNUM FROM ORDERS O, CUSTOMER C
+> WHERE O.CUSTNUM = C.CUSTNUM AND STATE = "CALIFORNIA")
+> GROUP BY ORDERNUM;
>> VOLUME $VOL1.SALES;
>> SELECT ORDERNUM, SUM (QTY_ORDERED * PRICE)
+> FROM PARTS P, ODETAIL O
+> WHERE O.PARTNUM = P.PARTNUM AND ORDERNUM IN
+> (SELECT ORDERNUM FROM ORDERS WHERE CUSTNUM IN
+> (SELECT CUSTNUM FROM CUSTOMER
+> WHERE STATE = "CALIFORNIA" ) )
+> GROUP BY ORDERNUM;

```

- This SQLCI example selects the value in the AUDIT column of the FILES catalog table in the PERSNL catalog. By displaying this column, you can see whether a table is defined as audited (Y) or nonaudited (N).

```

>> SELECT AUDIT FROM PERSNL.FILES
+> WHERE FILENAME = "\SYS1.$VOL1.PERSNL.JOB";
AUDIT
-----
Y
--- 1 row(s) selected.

```

- This SQLCI example uses a table, T, that has columns C (of data type CHARACTER) and I (of data type INTEGER). There are n possible values of C, any of which can occur multiple times, but the values of I are unique. The query returns the percent distribution of any value of C across the entire table.

Sample Table T:

C	I
--	-
N1	1
N1	4
N1	6
N2	2
N3	3
N3	5

The SQLCI query and the result:

```

>> --Print the percents
>> SELECT Y.C, (100.0*COUNT(*))/(COUNT(DISTINCT X.I) *
+>                                     COUNT(DISTINCT X.I))
+> FROM T X, T Y
+> GROUP BY Y.C;
C      (EXPR)
--  -----
N1                                50.0
N2                                16.6

```

```
N3                      33.3
--- 3 row(s) selected.
```

The query joins table T with itself; X and Y are correlation names so that a query can compare one row of the table with every other row. The GROUP BY Y.C clause produces sets of n*v values for each distinct C value in which these are true:

n

is the number of rows in the table.

v

is the number of occurrences of a particular C value.

COUNT(DISTINCT X.I) is n for each group. The second term in the select list represents this equation:

$$(n*v)/(n*n) = (v/n)$$

The SELECT statement displays a distinct value of C together with its percentage distribution over the table.

SERIALWRITES File Attribute

SERIALWRITES is a Guardian file attribute that specifies whether to write data serially or in parallel to the two disk devices that make up a mirrored volume. SERIALWRITES applies to key-sequenced, relative, and entry-sequenced tables and to indexes.

```
{ SERIALWRITES | NO SERIALWRITES }
```

SERIALWRITES	Selects serial mirror writes
NO SERIALWRITES	Selects parallel mirror writes

The table default is NO SERIALWRITES.

The index default is its table's value at index creation.

Considerations—SERIALWRITES

- For audited tables and indexes, use NO SERIALWRITES. For nonaudited tables or indexes, weigh the value of performance versus data reliability. SERIALWRITES can degrade response time, but it can also improve the reliability of data not protected by TMF auditing.
- SERIALWRITES tells the system to write one data block at a time to a mirrored pair of disks. If a system failure occurs, the failure affects only one disk, so the system uses the good copy of the data block after the failure.

With parallel writes (NO SERIALWRITES), the system writes to both disks of a mirrored pair simultaneously; performance is improved, but a system failure can affect both copies of the block being written.

SET DEFINE Command

SET DEFINE is an SQLCI command that sets a value for one or more DEFINE attributes in the working attribute set. The working attribute set determines values for attributes you do not specify in an ADD DEFINE command. (SET DEFINE is similar to the TACL command SET DEFINE and the OSS command set_define.)

```
SET DEFINE { LIKE define } [, attr value ] ... ;
           { attr value }
```

LIKE *define*

specifies the name of an existing DEFINE to use as a model for the new values of the working attribute set, optionally modified by *attr value* pairs that follow the LIKE clause.

If you use the LIKE clause, you cannot specify the CLASS attribute.

attr value

is the name and value of a DEFINE attribute to add to the working attribute set. For information about DEFINE attributes, see [DEFINES](#) on page D-27.

Considerations—SET DEFINE

- The working attribute set consists of values for the attributes of the current class. Only one class of attributes can be in the working set at one time.

For example, if you use SET DEFINE to set attribute values for class TAPE and enter this command, the working set provides values for all attributes except FILEID:

```
ADD DEFINE =T, CLASS TAPE, FILEID BACKUP
```

- Attributes are set in the order they are specified. Because the CLASS attribute erases the working set, you should set the CLASS attribute first, then set values for the other attributes. When you include CLASS in a SET DEFINE command, you establish a new working attribute set in which each attribute has its initial setting.
- If the value of an attribute is a Guardian name or subvolume name, the name is expanded immediately using the current default node, volume, and subvolume.
- If an error occurs on SET DEFINE, the working attribute set is unchanged.
- An attribute value does not change until you reset it with the RESET DEFINE command or another SET DEFINE command.
- SET DEFINE checks that the value you enter is valid for the attribute you specify and that the attribute is valid for the current class. Attribute consistency is not checked until you issue an ADD DEFINE, ALTER DEFINE, or SHOW command.
- Use the SHOW DEFINE command to display the current working attribute set before you use a SET DEFINE command.

Example—SET DEFINE

This example, SET DEFINE establishes a working attribute set for CLASS CATALOG. The ADD DEFINE does not specify any attributes or attribute values, so the working attribute set is associated with the DEFINE. In this case, the =CAT DEFINE is mapped to subvolume \SYS1.\$VOL1.PERSNL:

```
SET DEFINE CLASS CATALOG, SUBVOL \SYS1.$VOL1.PERSNL;
ADD DEFINE =CAT;
```

SET DEFMODE Command

SET DEFMODE is an SQLCI command that enables or disables the use of DEFINES in the current SQLCI session. (SET DEFMODE is similar to the TACL command SET DEFMODE.)

```
SET DEFMODE { ON } ;
             { OFF }
```

ON

enables the use of DEFINES. If DEFMODE is ON, you can execute commands that contain DEFINE names and you can create, modify, or delete DEFINES, display information about DEFINES, and propagate existing DEFINES to any processes you start from the SQLCI session.

ON is the default.

OFF

disables the use of DEFINES. With DEFMODE OFF, you cannot execute commands that contain DEFINE names and you cannot add DEFINES or propagate existing DEFINES to another process. You can modify, delete, and display information about existing DEFINES.

Example—SET DEFMODE

This example enables the use of DEFINES:

```
>> SET DEFMODE ON;
```

SET LAYOUT Command

SET LAYOUT is an SQLCI report writer command that sets layout options. Layout options affect the way a report appears on the screen or printed page.

```
SET [ LAYOUT ] option [ , option ] ... ;
```

option is:

```
{ CENTER_REPORT      { OFF | ON }
{ LEFT_MARGIN        number
{ LINE_SPACING       number
{ LOGICAL_FOLDING    { ON | OFF }
{ PAGE_COUNT         { number | ALL }
{ PAGE_LENGTH        { number | ALL }
{ RIGHT_MARGIN       number
{ SPACE              number
{ WINDOW             { TAB number | column }
```

You cannot specify the same option more than once in a single SET LAYOUT command.

For more information about a specific option, see the entry for that option.

Example—SET LAYOUT

This example sets the left margin for reports at byte position eight and also sets double spacing:

```
>> SET LAYOUT LEFT_MARGIN 8, LINE_SPACING 2;
```

SET PARAM Command

SET PARAM is an SQLCI command that sets values for parameters in your SQLCI session. SET PARAM overrides parameter values you set before entering SQLCI, but only for the duration of the SQLCI session.

```
SET [ PARAM ] ?param value [ , ?param value ]... ;

value is:
{ literal }
{ CURRENT_TIMESTAMP }
{ COMPUTE_TIMESTAMP (date) }
```

?param

is the name of the parameter to receive a value. An SQL parameter name is an SQL identifier preceded by a question mark.

literal

is a numeric or string literal, optionally enclosed in single or double quotation marks. Enclosing quotation marks are required only for string literals that include blank or comma characters; they make no difference otherwise. For example, this SET PARAM commands are equivalent:

```
SET PARAM ?NUM 9001, ?STR string;
SET PARAM ?NUM "9001", ?STR "string";
```

Within enclosing quotes of the same type, two quotes are handled as a single quote. For example, this SET PARAM commands are equivalent:

```
SET PARAM ?QUOTE " " " ";
SET PARAM ?QUOTE ' " ' ;
```

CURRENT_TIMESTAMP

is the Julian timestamp for the current date and time.

COMPUTE_TIMESTAMP (*date*)

is the Julian timestamp for the date and time you specify in *date* in this form:

```
{mm/dd/yyyy
{mm/dd/yyyy hh:nn:ss:mss:uss}}
```

yyyy Year, from 1 through 9999, 1 to 4 digits
mm Month, from 1 through 12, 1 to 2 digits
dd Day, from 1 through 31, 1 to 2 digits
hh Hour, from 0 through 23, 1 to 2 digits
nn Minute, from 0 through 59, 1 to 2 digits
ss Second, from 0 through 59, 1 to 2 digits
mss Millisecond, from 0 through 999, 1 to 3 digits
uss Microsecond, from 0 through 999, 1 to 3 digits

date cannot be an expression.

Considerations—SET PARAM

- SQL determines the data type for a parameter based on how you use the parameter in an SQL statement. As a result, a literal value you assign to a parameter might be interpreted differently in different SQL statements.

For example, the string “123” assigned to a parameter might be handled as an integer, a character string, or an invalid value—depending on whether you use the parameter in a statement that requires an integer, a character string, or a date-time value.

Similarly, a parameter that appears to have the value of a string literal with an associated character set, such as `_KANJI”c1c2”` (where “c1c2” indicates one double-byte character) is interpreted as having these value:

<code>c1c2</code>	if the data type is CHAR(2) CHAR SET KANJI
<code>_KANJI”c1c2”</code>	if the data type is CHAR(12)
<code>_K</code>	if the data type is CHAR(2)

For information about the way SQL determines data types for parameters, see [Parameters](#) on page P-11. For information about using the CAST function to specify a numeric or character data type for a parameter, see [CAST Function](#) on page C-4.

- You cannot use parameters in report formatting commands. A method for using TACL macros to pass parameters to report formatting commands is described in the *SQL/MP Report Writer Guide*.

Examples—SET PARAM

- In this example, the SELECT statement in the FINDSUP2 file finds suppliers of a specified part. The suppliers are located in a specified state.

```
VOLUME $VOL1.INVENT;
SELECT S.SUPPNUM, SUPPNAME FROM SUPPLIER S, PARTSUPP
WHERE S.SUPPNUM = PARTSUPP.SUPPNUM AND
PARTNUM = ?PN AND STATE = ?ST;
```

Before you can execute the SELECT statement, you must specify the state and part number with a SET PARAM command, as shown:

```
>> SET PARAM ?ST TEXAS, ?PN 4103;
>> OBEY FINDSUP2;
```

You do not have to enclose TEXAS in quotation marks because SQLCI determines from the STATE column definition that the column has a character data type.

- In these examples, the dates are in timestamp format and the OBEY command file INSORD contains these commands:

```
INSERT INTO SALES.ORDERS
VALUES (?ONUM, ?TODAY, ?DDATE, ?REP, ?CUSTNUM);
```

To insert an order, set the order date to the current day and enter a specific delivery date using the date conversion functions. The numbers of the order, sales representative, and customer are entered as numeric literals:

```
>> SET PARAM ?ONUM 800661, ?TODAY CURRENT_TIMESTAMP,
+> ?DDATE COMPUTE_TIMESTAMP ( 5/23/1988 ),
+> ?REP 221, ?CUSTNUM 7654;
>> OBEY INSORD;
```

These values are inserted in ORDERS:

```
ORDERNUM      800661
ORDER_DATE    (timestamp for current date and time)
DELIV_DATE    (timestamp for 5/23/88)
SALESREP      221
CUSTNUM       7654
```

Suppose that you have set values for some of the parameters of the previous INSERT command using the PARAM command before starting SQLCI:

```
4> PARAM ONUM 400410, CUSTNUM 7654
5> SQLCI
```

Before executing the INSORD OBEY command file, you change the value of the order number parameter and set values for the other parameters of the INSERT command:

```
>> SET PARAM ?ONUM 600480, ?TODAY CURRENT_TIMESTAMP,
+> ?DDATE COMPUTE_TIMESTAMP (7/5/1988), ?REP 221;
>> OBEY INSORD;
```

These values are inserted in ORDERS:

ORDERNUM	600480
ORDER_DATE	(timestamp for current date and time)
DELIV_DATE	(timestamp for 7/5/88)
SALESREP	221
CUSTNUM	7654

After setting a different order number and customer number, you execute the OBEY command file again:

```
>> SET PARAM ?ONUM 600481, ?CUSTNUM 123;
>> OBEY INSORD;
```

These values are inserted into ORDERS:

ORDERNUM	600481
ORDER_DATE	(timestamp for current date and time)
DELIV_DATE	(timestamp for 7/5/88)
SALESREP	221
CUSTNUM	123

When you exit SQLCI, the values of ?ONUM and ?CUSTNUM are 400410 and 7654, respectively, as set initially by the PARAM command.

SET SESSION Command

SET SESSION is an SQLCI command that sets session options for your SQLCI session.

```
SET [ SESSION ] option [ , option ] ... ;
```

option is:

{	AUTOWORK		[ON [AUDITONLY]		OFF]	}
	BREAK_KEY		{	OFF		ON	}	
	DISPLAY_ERROR		{	MAIN		ALL	}	
	ERROR_ABORT		{	OFF		ON	}	
	ERROR_TEXT		{	DETAIL		BRIEF	}	
	LIST_COUNT		{	<i>number</i>		ALL	}	
	MANDATORY_REPORT		{	OFF		ON	}	
	STATISTICS		{	OFF		ON	}	
	WARNINGS		{	OFF		ON	}	
	WRAP		{	OFF		ON	}	

You cannot specify the same option more than once in a single SET SESSION command.

AUTOWORK [ON [AUDITONLY] | OFF]

specifies whether SQLCI should automatically initiate a TMF transaction when you enter a DML command and how locking works in relation to nonaudited objects locked during the transaction.

ON [AUDITONLY]

directs SQLCI to start a TMF transaction when you enter a DML command, to commit the transaction if the command terminates successfully or to roll back the transaction if the command does not complete successfully, and to release locks on objects after execution of the command.

If you specify AUDITONLY, SQLCI does not release locks on nonaudited objects when the transaction is committed or rolled back.

The default when you start SQLCI is AUTOWORK ON (without AUDITONLY).

OFF

directs SQLCI not to start transactions automatically.

```
BREAK_KEY { OFF | ON }
```

specifies the SQLCI action when you press the Break key while executing an SQL command or an OBEY command.

OFF Return control to the previous Break key owner (type PAUSE to resume SQLCI later)

ON Retain control (the IN file must be a terminal)

Use OFF if you want to prevent someone from interrupting commands at your terminal.

The default when you start SQLCI is BREAK_KEY ON.

```
DISPLAY_ERROR { MAIN }
                { ALL }
```

controls which errors SQLCI displays when you enter an ERROR command. The setting you specify stays in effect until you set or reset it, or until you end your SQLCI session.

MAIN Display first error from most recent command only

ALL Display all errors and warnings from the command

The default when you start SQLCI is DISPLAY_ERROR ALL.

```
ERROR_ABORT { OFF | ON }
```

specifies the SQLCI action when an error occurs in a non-interactive SQLCI session or nested OBEY file. The main level of command is not affected when processing an interactive SQLCI session except for the completing code settings:

OFF Continue after error

ON Terminate immediately when an error is encountered

The default when you start SQLCI is ERROR_ABORT OFF.

ERROR_ABORT can be toggled between the ON and OFF settings within a session as required:

Session Type	ERROR_ABORT OFF	ERROR_ABORT ON
Interactive	OK=0	OK=0
Session	Warning=0	Warning=0
	Error=0	OK=0
Noninteractive	OK=0	OK=0
Session	Warning=1	Warning=1
	Error=2	Error =abend = 5

`ERROR_TEXT { DETAIL | BRIEF }`

specifies the default information SQLCI displays when you enter an ERROR command. You can override the ERROR_TEXT option with the ERROR command.

DETAIL Display error text, cause, effect, and suggestions for recovery

BRIEF Display only the error text

The default when you start SQLCI is ERROR_TEXT DETAIL.

`LIST_COUNT { number | ALL }`

specifies how many rows of data from a SELECT command to display before a pause.

number Display *number* rows

ALL Display all the rows of the result table

The default when you start SQLCI is LIST_COUNT ALL.

`MANDATORY_REPORT { OFF | ON }`

specifies whether SQLCI should print a report even if the associated query returns zero rows. This option has no effect when the associated query returns one or more rows.

OFF Do not print a report when the query does not return any rows. This is the default.

ON Print a report even if the query does not return any rows. The report format uses current report settings. In addition, the report contains the message “--- No rows selected.” when the query does not return any rows.

The default when you start SQLCI is MANDATORY_REPORT OFF.

`STATISTICS { OFF | ON }`

specifies whether to display statistics after each DDL, DML, or DCL command executes.

OFF Do not display statistics

ON Display statistics

The default when you start SQLCI is STATISTICS OFF.

For a description of the statistics displayed, see [DISPLAY STATISTICS Command](#) on page D-52.

`WARNINGS { OFF | ON }`

specifies whether to display warning messages. (SQLCI always displays error messages.)

`OFF` Do not display warning messages

`ON` Display warning messages

The default when you start SQLCI is `WARNINGS ON`.

`WRAP { OFF | ON }`

specifies whether SQLCI output lines that exceed the output device width are truncated or continued on the next line.

`OFF` Continue (wrap around) to the next line

`ON` Truncate the line if it does not fit

The default when you start SQLCI is `WRAP ON`.

Output device widths are:

Terminal	80 bytes
Unstructured Disk File	80 bytes
Structured File	Record length of file
Edit File	80 bytes; 239 if <code>RIGHT_MARGIN > 80</code>
Process (for example, Spooler collector)	132 bytes; 255 if <code>RIGHT_MARGIN > 132</code>
Printer	132 bytes; 255 if <code>RIGHT_MARGIN > 132</code>

Considerations—SET SESSION

- You can set up your SQLCI session environment by including the *sqlci-command* option when you start SQLCI. For example, you might create an OBEY command file named `PROFILE` that contains these commands:

```
VOLUME $VOL1.SALES;
CATALOG SALES;
SET SESSION AUTOWORK OFF, BREAK_KEY OFF;
LOG SQLCILOG;
```

Then you can prepare your environment quickly by entering this command:

```
SQLCI OBEY PROFILE;
```

This starts an SQLCI session, sets the current default volume and subvolume to `$VOL1.SALES`, sets the current catalog to `SALES`, turns off the `AUTOWORK` and `BREAK_KEY` session options, and begins logging session input and output to the file `SQLCILOG`.

- SQLCI responds to the Break key based on the setting of the BREAK_KEY session option:

Executing Command	BREAK_KEY ON	BREAK_KEY OFF	BREAK_KEY ON or OFF
SQL commands	Command rolls back; SQLCI prompts.	Control returns to previous owner.	
OBEY command	Command terminates; SQLCI prompts.	Control returns to previous owner.	
FC command			Command terminates and SQLCI prompts; BREAK_KEY has no effect.
Other SQLCI commands (whether or not within user-defined transaction)			Command terminates; control returns to previous owner; BREAK_KEY has no effect.

The previous owner is usually the process from which you started the SQLCI session. You can resume execution of the SQL or OBEY command.

An SQL command is a DDL, DML, DCL, PREPARE or EXECUTE command. If one of these commands is rolled back, SQLCI uses ROLLBACK WORK to terminate the current transaction and displays a message. The compiled form of the DDL or DML command is discarded. The next time you execute the command, SQLCI prepares it again and consequently causes a delay.

A DDL command that is not executing in a user-defined TMF transaction might complete before you press the Break key. To determine whether the command completed, you must examine the database. For example, you can use the FILES command to determine whether an object you were dropping when you pressed the Break key still exists. If you enter STOP *sqlci-process-id* at the command interpreter prompt, TMF rolls back the current TMF transaction. Changes to audited objects are undone, but changes to nonaudited objects are not undone.

- An interactive SQLCI does not terminate after an error in the main level of input despite ERROR_ABORT being set to ON. The main level of SQLCI input is the file specified by the SQLCI run command IN, INV, or INLINE options, or implied by their absence.

Interactive processing includes:

- A user's home terminal TACL process
- SQLCI / IN *file*, OUT *file*,.../

where *file* is an interactive device; typically a terminal (device type 6) or a process (device type 0)

- SQLCI /INLINE/
- SQLCI /INV *variable*, OUTV *variable*/

The use of TACL INLINE and INV/OUTV are special cases of a process device type classification.

Non-interactive processing includes:

- SQLCI /IN *file*,.../

where *file* is any disk file or non-interactive device

- Any nested OBEY file

Examples—SET SESSION

- To set the AUTOWORK option to OFF, enter:

```
>> SET AUTOWORK OFF;
```

To set AUTOWORK to ON and specify the AUDITONLY option, enter:

```
>> SET AUTOWORK ON AUDITONLY;
```

- To protect an update operation, you can disable the Break key before running the OBEY command file ORDUPDT. The commands in this file update the ORDERS table. If you press the Break key, SQLCI returns control to the command interpreter until you type PAUSE to resume your operation.

```
>> SET BREAK_KEY OFF;
>> OBEY ORDUPDT;
(Break key is pressed.)
5> PAUSE
```

- Suppose that the BREAK_KEY option is ON. During a TMF transaction that updates the price of each part in the PARTS table by 5 percent, you press the Break key. SQLCI rolls back the transaction.

```
>> SET BREAK_KEY ON;
>> BEGIN WORK;
>> UPDATE INVENT.PARTS SET PRICE = PRICE * 1.05;
(Break key is pressed.)
*** ERROR [10088] Command terminated by BREAK
```

- To set the DISPLAY_ERROR option to MAIN, enter:

```
>> SET DISPLAY_ERROR MAIN;
```

- To set the ERROR_TEXT option to BRIEF, enter:

```
>> SET ERROR_TEXT BRIEF;
```

To direct SQLCI to display ten rows at a time, enter:

```
>> SET LIST_COUNT 10;
>>
```

To enable the display of statistics and disable continuation of text to the next line, enter:

```
>> SET STATISTICS ON, WRAP OFF;
```

SET STYLE Command

SET STYLE is an SQLCI report writer command that sets style options. Style options affect the appearance of specific report items, such as underlines, headings, and date and time formats.

```
SET [ STYLE ] option [ , option ] ... ;
```

option is:

{ DATE_FORMAT	<i>date-format</i>	}
{ DECIMAL_POINT	{ "." "," }	}
{ HEADINGS	{ ON OFF }	}
{ NEWLINE_CHAR	"character"	}
{ NULL_DISPLAY	"character"	}
{ OVERFLOW_CHAR	"character"	}
{ ROWCOUNT	{ ON OFF }	}
{ SUBTOTAL_LABEL	"label"	}
{ TIME_FORMAT	<i>time-format</i>	}
{ UNDERLINE_CHAR	"character"	}
{ VARCHAR_WIDTH	<i>number</i>	}

The default settings are:

DATE_FORMAT	M2/D2/Y2
DECIMAL_POINT	period (.)
HEADINGS	ON
NEWLINE_CHAR	slash (/)
NULL_DISPLAY	question mark (?)
OVERFLOW_CHAR	asterisk (*)
ROWCOUNT	ON
SUBTOTAL_LABEL	*
TIME_FORMAT	HP2:M2:S2
UNDERLINE_CHAR	hyphen (-)
VARCHAR_WIDTH	80

* resets all style options to their default settings.

For information about a specific option, see the entry for that option.

Consideration—SET STYLE

You cannot specify the same option more than once in a single SET STYLE command.

Example—SET STYLE

This example activates report headings and specifies an asterisk as the character to display for null values:

```
>> SET STYLE HEADINGS ON, NULL_DISPLAY "*";
```

SETSCALE Function

SETSCALE is a function that specifies the scale of a host variable to SQL. You can use SETSCALE in SQL statements in C, Pascal, or TAL programs.

You use SETSCALE to store scaled values (such as prices) in a database, retrieve database values into host variables in the program, or reference values in the database for comparison operations. The scale information is valid only in the context of the SQL statement; the program must handle scaling for host language statements.

```
SETSCALE ( :host-var [[INDICATOR] :ind-var], scale )
```

:host-var

is an integer variable in a host language program.

[INDICATOR] :ind-var

specifies an indicator variable associated with the host variable.

scale

is an integer that specifies the scale of *host-var*. The values allowed depend on the size of *host-var*:

2-byte integers	0 through 5 decimal digits
4-byte integers	0 through 10 decimal digits
8-byte integers	0 through 18 decimal digits

Consideration—SETSCALE

SETSCALE directs SQL to use *host-var* in the context of SQL statements as if *host-var* were declared with a scale of *scale*.

If the value in *host-var* is entered into the database through an INSERT or UPDATE, the host language program must assign a value that allows for the scale to *host-var*.

For example, if the program is representing a price of \$123.45, the program should assign 12345 to *host-var* and use SETSCALE to specify a scale of two.

If the value is being retrieved from the database through a SELECT operation, SQL assigns a value that allows for the scale to *host-var*. For example, if SQL is storing 123.45, the value 12345 is returned to *host-var* when the program specifies SETSCALE with a scale of two in the SELECT statement.

To use SETSCALE in an expression, you must apply SETSCALE to each operand individually rather than to the result of the expression. For example, this expression adds two prices with a scale of two decimal places:

```
SETSCALE (:PRICE1, 2) + SETSCALE (:PRICE2, 2)
```

Examples—SETSCALE

- This C program fragment uses SETSCALE with an INSERT to create a new row with the value 98.34 in the PARTS.PRICE column after storing the value in host variable :HV1. The value is multiplied by 100 for storing as a whole number.

```
HV1 = 9834;
EXEC SQL  INSERT INTO =PARTS (PRICE)
VALUES ( SETSCALE (:hv1, 2) );
```

- This C program fragment uses SETSCALE with UPDATE to change a value in the PARTS.PRICE column to \$158.34. The value is multiplied by 100 and stored in host variable :HV2.

```
HV2 = 15834;
EXEC SQL  UPDATE PARTS SET PARTS.PRICE = SETSCALE (:hv2, 2)
WHERE PARTS.PARTDESC = "DISK CONTROLLER";
```

- This C program fragment uses SETSCALE with SELECT to retrieve the value for a disk controller from the PARTS.PRICE column and stores the value in host variable :HV3. The value has a scale of two.

```
EXEC SQL  SELECT PARTS.PRICE INTO SETSCALE ( :HV3, 2 )
FROM =PARTS
WHERE PARTS.PARTDESC = "DISK CONTROLLER";
```

- This C program fragment uses SETSCALE with SELECT to retrieve the part description for the part with a price of \$999.50. The price value is stored in host variable :HV4 and supplied to SQL in the search condition. The retrieved value is stored in host variable :HVSTORE.

```
HV4 = 99950;
EXEC SQL  SELECT PARTS.PARTDESC INTO :HVSTORE
FROM =PARTS
WHERE PARTS.PRICE = SETSCALE ( :hv4, 2 );
```

Shorthand View

A shorthand view is a view derived from one or more tables or views by joining tables or views, by projecting columns, by restricting rows, or by a combination of these actions. Shorthand views can be read, but cannot be updated or secured.

SHOW CONTROL Command

SHOW CONTROL is an SQLCI command that displays the current values of options set by the CONTROL EXECUTOR, CONTROL QUERY, and CONTROL TABLE statements.

```
SHOW CONTROL ;
```

Example—SHOW CONTROL

```
>> SHOW CONTROL;
-----
Current Environment
-----
CONTROL TABLE ORDERS TABLELOCK OFF
CONTROL TABLE ORDERS TIMEOUT 10.00 SECONDS
CONTROL TABLE ORDERS SEQUENTIAL UPDATE ON
CONTROL TABLE ORDERS SYNCDEPTH 1
CONTROL TABLE ORDERS MDAM ON
CONTROL TABLE ORDERS MDAM USE 3 KEY COLUMNS
CONTROL TABLE ORDERS MDAM ACCESS DENSE
```

SHOW DEFINE Command

SHOW DEFINE is an SQLCI command that displays all or part of the working attributes set. (SHOW DEFINE is similar to the TACL command SHOW DEFINE and the OSS command show_define.)

```
SHOW DEFINE [ attr ] ;
            [ *      ]
```

attr

displays the attribute you specify and its value. You can specify any attribute that is legal for the current class. (If you do not know the current class, specify CLASS.)

*

displays all attributes in the working attribute set and their values. Optional attributes that have no current value are not displayed.

Consideration—SHOW DEFINE

If you do not specify an attribute or an asterisk, SHOW DEFINE displays all attributes with a current value and warns you to specify values for required attributes without current values.

Attributes whose values violate consistency rules are flagged with an asterisk (*).

Examples—SHOW DEFINE

- This example displays the CLASS attribute from the working attributes set:

```
>>SHOW DEFINE CLASS;
CLASS                                MAP
```

- This example displays the current set of working attributes. In the example, no value is currently assigned to the FILE attribute. You must supply a value for the FILE attribute to add a DEFINE because the working attribute set does not provide a default value.

```
>>SHOW DEFINE *;
CLASS                                MAP
FILE                                ??
```

Current attribute set is incomplete

SHOW DEFMODE Command

SHOW DEFMODE is an SQLCI command that displays the current DEFMODE setting. (SHOW DEFMODE is similar to the TACL command SHOW DEFMODE.) DEFMODE is an attribute of a process that controls whether you can create DEFINES from the process and whether DEFINES are propagated when the process starts another process. For more information, see [DEFINES](#) on page D-27.

```
SHOW DEFMODE ;
```

Example—SHOW DEFMODE

This example displays the current DEFMODE:

```
>>SHOW DEFMODE ;
Defmode                ON
```

SHOW LAYOUT Command

SHOW LAYOUT is an SQLCI report writer command that displays the values of the current layout options. Layout options affect the way a report appears on a terminal screen or printed page.

```
SHOW { [ LAYOUT ] option [, option] ... } ;
      { LAYOUT * }
```

option is:

```
{ CENTER_REPORT      }
{ LEFT_MARGIN        }
{ LOGICAL_FOLDING    }
{ PAGE_COUNT         }
{ PAGE_LENGTH        }
{ RIGHT_MARGIN       }
{ SPACE              }
{ WINDOW             }
```

* displays all layout options.

For information about a specific option, see the entry for that option.

Example—SHOW LAYOUT

This example displays the current margin and line-spacing values:

```
>> SHOW LAYOUT LEFT_MARGIN, RIGHT_MARGIN, LINE_SPACING;
LEFT_MARGIN      5
RIGHT_MARGIN     80
LINE_SPACING     2
```

SHOW PARAM Command

SHOW PARAM is an SQLCI command that displays the current parameter values.

```
SHOW { [ PARAM ] name [ , name ] ... } ;
      { PARAM * }
```

name

identifies the parameter you want to display.

*

displays the values of all parameters.

Examples—SHOW PARAM

- This example displays two parameter values:

```
>> SHOW PARAM ?PNUM, ?ST;
?PNUM      4103
?ST        TEXAS
```

- This example displays the value of every currently defined parameter:

```
>> SHOW PARAM *;
```

SHOW PREPARED Command

SHOW PREPARED is an SQLCI command that displays prepared commands.

```
SHOW PREPARED { command-name [ , command-name ] ... } ;
               { *
```

command-name

is the name you specified for the command when you prepared it.

*

displays all the currently prepared commands.

Example—SHOW PREPARED

This example displays a prepared command named SELALLCU:

```
>> SHOW PREPARED SELALLCU;
select * from sales.customer
>>
```

SHOW REPORT Command

SHOW REPORT is an SQLCI report writer command that displays the current SELECT statement and the current report formatting commands.

```
SHOW REPORT { { command } [ , command ] ... } ;
```

command is:

```
{ BREAK
  BREAK FOOTING
  BREAK TITLE
  DETAIL
  NAME
  PAGE FOOTING
  PAGE TITLE
  REPORT FOOTING
  REPORT TITLE
  SELECT
  SUBTOTAL
  TOTAL }
```

* displays the SELECT and all report formatting commands.

For information about a specific command, see the entry for that command. (The command listed as BREAK refers to the BREAK ON command.)

Example—SHOW REPORT

This example displays the current report title and page footer:

```
>> SHOW REPORT TITLE, PAGE FOOTING;
REPORT TITLE "ORDER DETAILS" CENTER;
PAGE FOOTING TAB 50, "Page ", PAGE_NUMBER AS I3;
```

SHOW SESSION Command

SHOW SESSION is an SQLCI command that displays the values of the current session options.

```
SHOW [ SESSION ] { option [, option ] ... } ;
                  *
```

option is:

```
{ AUTOWORK
  BREAK_KEY
  DISPLAY_ERROR
  ERROR_ABORT
  ERROR_TEXT
  LIST_COUNT
  MANDATORY_REPORT
  STATISTICS
  WARNINGS
  WRAP }
```

* displays all session options.

For an explanation of each session option, see [SET SESSION Command](#) on page S-39.

Example—SHOW SESSION

This example displays the current values of all options:

```
>> SHOW SESSION *;
-----
Current SESSION Option Values
-----
AUTOWORK          ON
BREAK_KEY         OFF
DISPLAY_ERROR     MAIN
ERROR_ABORT       OFF
ERROR_TEXT        DETAIL
LIST_COUNT        10
MANDATORY_REPORT  ON
STATISTICS        OFF
WARNINGS          ON
WRAP              OFF
```

SHOW STYLE Command

SHOW STYLE is an SQLCI report writer command that displays the current style options. Style options affect the appearance of specific report items, such as underlines, headings, and date and time formats.

```
SHOW { [ STYLE ] option [, option ] ... } ;
      { STYLE *
      }

option is:

      { DATE_FORMAT      }
      { DECIMAL_POINT    }
      { HEADINGS          }
      { NEWLINE_CHAR      }
      { NULL_DISPLAY      }
      { OVERFLOW_CHAR     }
      { ROWCOUNT         }
      { SUBTOTAL_LABEL    }
      { TIME_FORMAT       }
      { UNDERLINE_CHAR    }
      { VARCHAR_WIDTH     }
```

* displays all style options.

For information about a specific option, see the entry for that option.

Example—SHOW STYLE

This example displays the current setting for three style options:

```
>> SHOW STYLE NEWLINE_CHAR, OVERFLOW_CHAR, UNDERLINE_CHAR;
NEWLINE_CHAR      @
OVERFLOW_CHAR     *
UNDERLINE_CHAR    _
```

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 is also an operable plan for the other. (SQL does not perform similarity checks for objects referenced in parallel execution plans.)

You can reduce recompilation 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.

Similarity checks work by comparing information stored in a program file at explicit compilation time with information current at recompilation time.

During an explicit recompilation, the SQL compiler uses similarity checks to differentiate between invalid and inoperable plans if you specify **COMPILE INOPERABLE PLANS**.

During an automatic recompilation, the SQL executor uses similarity checks to differentiate between invalid and inoperable plans if you specified **CHECK INOPERABLE PLANS** when you last explicitly compiled the program.

General Rules for Similarity

Only tables, protection views, and collations can be similar. A table can be similar to another table, a protection view can be similar to another protection view, and a collation can be similar to another collation. No object other than a table, protection view, or collation is ever considered similar to another object (even an object of the same type) for the purposes of a similarity check.

A table or protection view compared to a compile-time table or protection view in a similarity check must have its **SIMILARITY CHECK** option set to **ENABLE**. (When you create a table or view, the default is **SIMILARITY CHECK DISABLE**, but you can specify **SIMILARITY CHECK ENABLE** with the **CREATE TABLE**, **CREATE VIEW**, **ALTER TABLE**, or **ALTER VIEW** statements.)

The value of the **SIMILARITY CHECK** option for a table or protection view affects only that table or view, not views defined on that table or view.

Similarity Between Protection Views

A protection view referenced at execution time (or recompilation time) is similar to the protection view used at compilation time if the protection view referenced at execution time has the **SIMILARITY CHECK** option enabled and if both protection views

- Have a similar underlying table
- Project the same columns from the underlying table
- Have the same column names
- Have the same selection expression (determined by a binary comparison of the parsed, internal representation of the two selection expressions)

Similarity Between Tables

A table referenced at execution time (or recompilation time) is similar to the table used at compilation time if the table referenced at execution time has the **SIMILARITY CHECK** option enabled and if the tables are the same, except in these aspects:

- These characteristics can differ:
 - Table names and table statistics

- Data within the tables
- Column headings and help text
- Catalog where the table is registered
- Comments on columns, constraints, indexes, or tables
- Number of partitions and partitioning key ranges
- Key tags (or values) for indexes
- Creation and redefinition timestamps

- These file attributes can differ:

ALLOCATE	LOCKLENGTH	SECURE
AUDITCOMPRESS	MAXEXTENTS	SERIALWRITES
BUFFERED	NOPURGEUNTIL	TABLECODE
CLEARONPURGE	OWNER	VERIFIEDWRITES
EXTENT (primary and secondary)		

The AUDIT file attribute can also differ unless the program includes a statement that performs a DELETE or UPDATE set operation on a nonaudited table with a SYNCDEPTH of 1.

- The number of indexes can differ, but all indexes used in the execution plan must exist for both tables.
- The table referenced at execution time can have more columns than the table referenced at compile time if the common columns have identical attributes and if the statement that references the table is not one of these:
 - An INSERT statement that does not explicitly specify names of columns into which to insert values.
 - A statement that uses unqualified column names in a way that becomes ambiguous because of the additional columns.
 - A statement that uses a SELECT list containing an asterisk, unless the asterisk is of the form COUNT (*).

Tables referenced in statements such as these can be similar:

```
SELECT COUNT (*) FROM tablea;
SELECT columnx FROM tablea WHERE
  columny relational-op (SELECT COUNT(*) FROM tableb);
```


Tables referenced in the position of *tablea* in this statement can be similar, but tables referenced in the position of *tableb* in this statements can never be similar:

```
SELECT columnx FROM tablea
WHERE EXISTS SELECT [DISTINCT] * FROM tableb;
SELECT tableb.*, tablea.x FROM tableb, tablea;
```

Similarity Between Collations

Collations that are explicitly referenced in an SQL query are similar only if they are equal. SQL uses the procedure `CPRL_COMPAREOBJECTS_` to determine equivalence of collations.

Displaying the Similarity Check Attribute

Information on the similarity check for a table can be retrieved from the catalog where the table is registered. To display this information, enter This SQLCI query:

```
Select tablename,similaritycheck from catalog.tables
where tablename like "% TABLENAME%";
```

TABLENAME must be uppercase.

SLACK File Attribute

SLACK is a file attribute that specifies the minimum percentage of space to leave for future insertions when loading data and index blocks. SLACK applies only to key-sequenced files and to indexes.

<i>SLACK percent</i>

percent

is an integer from 0 to 99 that specifies the percent of empty space to leave in each data and index block during loading.

The default is 15 percent.

Purpose of SLACK

- SLACK specifications are usually between 15 and 25 percent.
- Specifying a larger than normal SLACK value when a file is initially loaded and many more inserts are expected can improve performance by reducing the number of block splits required when inserts occur.
- For a file expected to have little activity, you can save disk space by specifying a smaller than normal SLACK value.

SPACE Option

SPACE is an option of the SQLCI report writer SET LAYOUT command that specifies the default number of spaces between columns of report print items. Each space occupies a single byte position, regardless of the character set in use.

SPACE *number*

number

is an integer in the range 0 through 255 that specifies the default number of spaces between print items.

The default is 2.

Considerations—SPACE

- If you specify the SPACE clause on the DETAIL command, the value you specify overrides the value of the SPACE layout option for that detail line.
- SQL does not print spaces before or after a string literal in a report unless you specify a heading for the report column that includes the string literal. If you specify a heading, SQL prints the default number of spaces before and after the string literal.

Examples—SPACE

- This example demonstrates that SQL does not print spaces before or after a string literal unless the report column that includes the string literal has a heading:

```
>> SET LIST_COUNT 0;
>> SELECT * FROM PERSONL.JOB;
S> DETAIL JOBCODE, "****", JOBDESC;
S> LIST NEXT 4;
JOBCODE      JOBDESC
-----
100****MANAGER
200****PRODUCTION SUPV
250****ASSEMBLER
300****SALESREP
S> DETAIL JOBCODE, "****" HEADING "      ", JOBDESC;
S> LIST FIRST 4;
JOBCODE      JOBDESC
-----
100   ***   MANAGER
200   ***   PRODUCTION SUPV
250   ***   ASSEMBLER
300   ***   SALESREP
```

- This example uses the SPACE layout option to change the default number of spaces between columns. The content of the report (not the spacing) is the same as in the previous example.

```
>> SET LAYOUT SPACE 5;
>> SET LIST_COUNT 0;
>> SELECT * FROM PERSONL.JOB;
S> LIST NEXT 4;
JOBCODE      JOBDESC
-----
      100      MANAGER
      200      PRODUCTION SUPV
      250      ASSEMBLER
      300      SALESREP
```

SQL Directive

The SQL directive indicates to a host language compiler that a program contains SQL statements.

For more information about the SQL directive, see the SQL/MP programming manual for your host language.

SQL Identifiers

An SQL identifier can contain up to 30 letters (A through Z or a through z), digits (0 through 9), or underscore (_) characters. The first character must be a letter. SQL ignores case in SQL identifiers; for example, employee and EMPLOYEE are equivalent.

SQL identifiers are used as column, constraint, correlation, cursor, parameter, and statement names and cannot be the same as any keyword reserved for SQL statements. For a list of reserved SQL keywords, see [Reserved Words](#) on page R-11.

Valid SQL identifiers are:

- ColumnName
- MONTH3ORDERS
- second_low_value
- EMPLOYEE
- ORD_DATE
- THE_END_

SQLCI

SQLCI, the SQL/MP conversational interface, executes commands and SQL statements entered interactively at a terminal or through a command file. SQLCI is useful for ad hoc queries and reports, for testing SQL statements before adding them to programs, for comparing the relative efficiency of different versions of a query, and for database administration tasks such as defining and modifying the structure of the database.

SQLCI provides many commands that are not available in embedded SQL programs but also accepts almost any statement that you can include in an embedded SQL program. A few SQL statements have options that are not allowed in SQLCI (or options that are allowed only in SQLCI); these are noted in the description of the statement.

For a summary of commands unique to SQLCI, see [SQLCI Commands](#) on page S-63, and [Report Writer](#) on page R-7. For a list of SQL statements you can use in programs or in an SQLCI session, see [Statements](#) on page S-73.

An SQLCI Session

You start an SQLCI session with the command interpreter SQLCI command and end an SQLCI session with an EXIT command. During a session, SQLCI prompts you to enter SQLCI statements or SQLCI commands with one of these prompts:

- >> The standard prompt. Enter any SQLCI command or SQL statement except CANCEL, LIST, or a report formatting command.
- +> The continuation prompt. Continue the SQLCI command or SQL statement from the previous line or enter a semicolon to end it.
- S> The select-in-progress prompt. Enter a LIST, CANCEL, report formatting command, or one of the other SQLCI commands allowed while a SELECT is in progress. You can also press RETURN, which is equivalent to LIST 1.
- D> The dedicated-operation-in-progress prompt. Enter a CONTINUE statement to commit or roll back the operation, or enter any SQLCI command except ADD DEFINE, ALTER DEFINE, CATALOG, DELETE DEFINE, EXIT, SET DEFMODE, SYSTEM, VOLUME, or a utility command or DDL statement on the object of the dedicated operation.
- .. The FC command prompt. For information, see [Considerations—FC](#) on page F-1.

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

Environment	The files to use for SQLCI output, logging, and reports; the default node, volume, subvolume, and catalog
DEFINEs	DEFINEs that associate an actual file or object with a logical name used in a command or statement
Session Options	Options set by the SET SESSION command that control the response to the Break key, automatic creation of TMF transactions, the amount of SELECT information displayed before a pause, the display of execution statistics for DML statements, the display of errors and warnings, and the wrapping of text
Layout Options	Options set by the SET LAYOUT command that define the layout of information displayed after a SELECT
Style Options	Options set by the SET STYLE command that determine the style of report elements such as date and time formats and the underline character
Parameters	Parameter values set by the SET PARAM command to substitute for parameter names when a command or statement executes
Prepared Commands	Commands compiled during an SQLCI session for execution later in the session
Report Commands	Report formatting commands and the most recent SELECT command entered. SQLCI saves these commands automatically during your session. You can display, delete, or fix and reexecute these commands with SHOW, RESET, or FC.

The SQLCI Command

The SQLCI command is a TACL implied run command that starts an SQLCI session. In the OSS environment, you can invoke the SQLCI command by preceding it with the gtac command. For more information about SQLCI and OSS, see [Considerations—SQLCI](#) on page S-62.

```
SQLCI [ / [ IN in-file ] [, OUT list-file ] [, NOWAIT ]
      [, run-option ] ... / ] [ sqlci-command ; ] ...
```

IN *in-file*

specifies the file from which SQLCI reads your commands; *in-file* is a valid Guardian name for a local or remote file. You can specify a disk file, a device such as a terminal, or a process. If you do not specify an IN file, SQLCI uses the same IN file as the command interpreter—usually the home terminal.

OUT *list-file*

specifies the file to which SQLCI writes or displays prompts, messages, listings of data, and so forth; *list-file* is a valid Guardian name for a local or remote file. You can specify a disk file, terminal, process, magnetic tape, line printer, or spooler collector. If you do not specify an OUT file, SQLCI uses the same OUT file as the command interpreter—usually the home terminal.

If an existing file contains data, SQLCI appends its output to the existing information.

NOWAIT

specifies that the command interpreter should not wait while SQLCI runs, but it should return a command input prompt after sending the startup message to SQLCI. If you omit this option, the command interpreter pauses while SQLCI runs.

With NOWAIT, you should include SET BREAK_KEY OFF as an *sqlci-command* option. This strategy prevents SQLCI from taking control from another process when you press the Break key.

run-option

is an option of the RUN command, which is described in the *TACL Reference Manual*.

sqlci-command

is any SQLCI command. You must terminate each SQLCI command with a semicolon (;), as shown in the command description. After executing the commands you enter with this option, SQLCI starts reading from the IN file.

Considerations—SQLCI

- TMF must be enabled while you run SQLCI.
- In the OSS environment, you can start an SQLCI session by using the gtacl command. For example, this command begins an interactive SQLCI session:

```
gtacl -p sqlci
```

To end the SQLCI session and return to the OSS prompt, use the SQLCI EXIT command. You can also start an SQLCI session, execute SQLCI commands, and return to the OSS prompt by using the gtacl command with the -c option, as in this example:

```
gtacl -c "sqlci; fileinfo; exit;"
```

For more information about using the gtacl command in the OSS environment, see the *Open System Services Shell and Utilities Reference Manual* or enter this command at the OSS prompt:

```
man gtacl
```

- You can set up your SQLCI session environment by including the *sqlci-command* option when you start SQLCI. For example, you might create an OBEY command file named PROFILE that contains these commands:

```
VOLUME $VOL1.SALES;
CATALOG SALES;
SET SESSION AUTOWORK OFF, BREAK_KEY OFF;
LOG SQLCILOG;
```

Then you can prepare your environment quickly by entering the:

```
SQLCI OBEY PROFILE;
```

This command starts an SQLCI session, sets the current default volume and subvolume to \$VOL1.SALES, sets the current catalog to SALES, turns off the AUTOWORK and BREAK_KEY session options, and begins logging session input and output to the file SQLCILOG.

Example—SQLCI

This is an example of a simple SQLCI session started from a TACL prompt to set up tables for testing. Commands entered by the user are shown in lowercase.

```
157>SQLCI
SQL Conversational Interface - T9191D30
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1987-1994
>>volume mycat;
>>create table test1 (a int, b char (3));
--- SQL operation complete.
>>create table test2 (x datetime year to minute,
+>                    y national character varying (30),
+>                    z int, primary key z);
--- SQL operation complete.
>>exit
```

SQLCI Commands

SQLCI commands are free-format: spaces are not significant except within character strings or numbers. Commands can contain up to 132 characters per line and can be in uppercase, lowercase, or mixed case.

Each SQLCI command (in addition to each SQL statement entered through SQLCI) must be terminated by a semicolon. You can include several commands on the same line as long as each one is terminated by a semicolon.

You can continue any command over multiple lines, breaking that command at any point except within a word, a numeric literal, or a multicharacter operator (for example, >=). Except when breaking a string literal, you do not have to enter a continuation symbol (&).

[Table S-2](#) on page S-64 summarizes the major SQLCI commands. (SQLCI commands used primarily for report writing are summarized separately in the entry REPORT

WRITER. SQL statements—most of which you can use in SQLCI—are summarized separately in the entry STATEMENTS.)

Table S-2. Summary of SQLCI Commands (page 1 of 4)

Command	Action
<u>CANCEL Command</u>	Cancels the current SELECT
<u>CATALOG Command</u>	Selects a different default catalog
<u>CLEANUP Command</u>	Purges damaged SQL objects
<u>CONVERT Command</u>	Generates commands to convert Enscribe files to SQL tables
<u>COPY Command</u>	Copies data into an Enscribe file or SQL table, adding to existing data
<u>CREATE SYSTEM CATALOG Command</u>	Creates the system catalog when NonStop SQL/MP is first installed
<u>DELETE DEFINE Command</u>	Deletes DEFINES
<u>DISPLAY STATISTICS Command</u>	Displays statistics for a recently compiled DML statement
<u>DISPLAY USE OF Command</u>	Displays a list of SQL objects that depend on a specified object
<u>DOWNGRADE CATALOG Command</u>	Downgrades catalogs to an older version
<u>DOWNGRADE SYSTEM CATALOG Command</u>	Downgrades the system catalog to an older version
<u>DROP SYSTEM CATALOG Command</u>	Deletes the system catalog
<u>DUP Command</u>	Duplicates files or objects
<u>EDIT Command</u>	Invokes the EDIT text editor
<u>ENV Command</u>	Displays attributes of the current SQLCI session
<u>ERROR Command</u>	Displays error text
<u>EXIT Command</u>	Ends an SQLCI session
<u>FC Command</u>	Edits and reexecutes a previously entered command
<u>FILEINFO Command</u>	Lists physical characteristics of collations, files, indexes, programs, tables, and views
<u>FILENAMES Command</u>	Displays names of files and tables that match a specified pattern
<u>FILES Command</u>	Displays names of files and tables on a subvolume
<u>FUP Command</u>	Invokes the File Utility Program
<u>HELP Command</u>	Displays information about NonStop SQL/MP

Table S-2. Summary of SQLCI Commands (page 2 of 4)

Command	Action
<u>HISTORY Command</u>	Displays recently executed commands
<u>INFO DEFINE Command</u>	Displays attributes and values associated with DEFINES
<u>INITIALIZE SQL Command</u>	Prepares a node to run SQL
<u>INVOKE Directive and Command</u>	Generates a record description of a table or view
<u>LIST Command</u>	Displays rows returned by a SELECT
<u>LOAD Command</u>	Loads data into an Enscribe file or SQL table, overwriting existing data
<u>LOG Command</u>	Starts or ends session logging
<u>MODIFY CATALOG</u>	Modifies node names in SQL catalogs on the local node
<u>MODIFY LABEL</u>	Modifies node numbers in file labels on the local node
<u>MODIFY REGISTER Command</u>	Registers a user-defined catalog in the local system catalog
<u>OBEY Command</u>	Executes commands from a file
<u>OUT Command</u>	Specifies or closes the output file
<u>PAGE FOOTING Command</u>	Specifies text for the bottom of each page
<u>PAGE TITLE Command</u>	Specifies text for page title
<u>PERUSE Command</u>	Invokes the PERUSE utility program
<u>PURGE Command</u>	Purges objects or files
<u>PURGEDATA Command</u>	Erases data from files or tables
<u>REPORT FOOTING Command</u>	Specifies text for the end of a report
<u>REPORT TITLE Command</u>	Specifies text for the main title of a report
<u>RESET DEFINE Command</u>	Clears DEFINE values
<u>RESET LAYOUT Command</u>	Resets layout options
<u>RESET PARAM Command</u>	Clears parameter values
<u>RESET PREPARED Command</u>	Resets prepared commands

Table S-2. Summary of SQLCI Commands (page 3 of 4)

Command	Action
<u>RESET REPORT Command</u>	Resets report formatting commands
<u>RESET SESSION Command</u>	Resets session options
<u>RESET STYLE Command</u>	Resets style options
<u>SAVE Command</u>	Writes values of the commands and session attributes to a file
<u>SECURE Command</u>	Changes the owner or security for a file or table
<u>SET DEFINE Command</u>	Sets a value for a DEFINE
<u>SET DEFMODE Command</u>	Enables or disables DEFINES
<u>SET LAYOUT Command</u>	Sets values for layout options
<u>SET PARAM Command</u>	Sets values for parameters referred to in SQLCI commands
<u>SET SESSION Command</u>	Sets session options
<u>SET STYLE Command</u>	Sets style options
<u>SHOW CONTROL Command</u>	Displays control options
<u>SHOW DEFINE Command</u>	Displays the set of working attributes for creating DEFINES
<u>SHOW DEFMODE Command</u>	Displays the current DEFMODE setting
<u>SHOW PARAM Command</u>	Displays parameter values
<u>SHOW PREPARED Command</u>	Displays prepared commands
<u>SHOW REPORT Command</u>	Displays report formatting commands and the most recent SELECT statement
<u>SHOW SESSION Command</u>	Displays the session options
<u>SHOW STYLE Command</u>	Displays current style options
<u>SYSTEM Command</u>	Changes the default node
<u>TEDIT Command</u>	Invokes the PS Text Edit text editor

Table S-2. Summary of SQLCI Commands (page 4 of 4)

Command	Action
UPGRADE CATALOG Command	Upgrades catalogs to a newer version of NonStop SQL/MP
UPGRADE SYSTEM CATALOG Command	Upgrades the system catalog to a newer version of NonStop SQL/MP
VERIFY Command	Checks the consistency and validity of object definitions in catalogs and file labels; lists invalid programs
VOLUME Command	Changes the default volume or subvolume
! Command	Reexecutes a previous command

For more information about a specific SQLCI command, see the entry for that command.

SQLCODE

SQLCODE is a variable in a host program to which SQL returns status information.

For more information about SQLCODE, see the SQL/MP programming manual for your host language.

SQLCOMP Command

The SQLCOMP command starts the SQL/MP compiler from a TACL process in the Guardian environment. You use SQLCOMP to SQL-compile Guardian host language programs that contain embedded SQL statements.

For information about SQLCOMP, see the SQL/MP programming manual for your host language.

Standards Conformance

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

The remainder of this entry summarizes the exceptions and extensions to the standard.

Exceptions to Conformance With Entry Level SQL 1992

This list summarizes the major ways in which NonStop SQL/MP does not conform to ANSI/ISO Entry Level SQL:

- Catalog, schema, table, view, and collation names

NonStop SQL/MP does not have the concept of a catalog or schema equivalent to those of ANSI/ISO SQL. However, the SQL/MP concept of a catalog is similar to that of an ANSI/ISO SQL schema in many respects.

SQL/MP table, view, and collation names differ from table, view, and collation names in ANSI/ISO SQL. ANSI/ISO SQL table, view, and collation names are of the form:

catalog.schema.table

SQL/MP table names are Guardian names, which allow a node and disk specification where ANSI/ISO SQL allows *catalog*. Guardian names also allow what NonStop SQL/MP calls a catalog where ANSI/ISO SQL allows *schema*.

- Security

The ANSI/ISO SQL security model differs overall from the Guardian security model (including the optional Safeguard security product) used by NonStop SQL/MP. Because of the differences in the model, NonStop SQL/MP does not implement table privileges (SELECT, INSERT, UPDATE, DELETE, and REFERENCES), column privileges (UPDATE and REFERENCES), or the USER value.

ANSI/ISO SQL defines authorization with the GRANT and REVOKE statements. NonStop SQL/MP uses the ALTER statements and the SECURE command.

- Constraints

ANSI/ISO SQL allows you to define a unique constraint when you create a table or to provide a UNIQUE qualifier on a column definition. To achieve a unique constraint in NonStop SQL/MP, you specify a unique column as the primary key of a table or create a unique index on a column.

NonStop SQL/MP does not allow you to qualify a constraint name with a catalog and schema name.

NonStop SQL/MP does not support the FOREIGN KEY table constraint or the REFERENCES column constraint.

- Views

ANSI/ISO SQL allows you to update a view if the query that defines the view can be updated. NonStop SQL/MP defines two types of views: protection views that can be updated and shorthand views that cannot be updated.

ANSI/ISO SQL allows you to create views on views. NonStop SQL/MP allows you to create views on shorthand views, but not on protection views.

ANSI/ISO SQL provides a SELECT DISTINCT clause on the CREATE VIEW statement and allows a WHERE clause in a view definition to refer to columns that are not in the select list. NonStop SQL/MP does not provide such a clause.

- Statement atomicity and transaction definition

ANSI/ISO SQL provides both statement atomicity (all changes made during a multirecord operation are canceled if an error occurs during the operation) and implicit transaction beginnings.

In NonStop SQL/MP, audited tables have equivalent protection for a TMF transaction but not for a single statement. You must explicitly begin each transaction within a program, and you must explicitly begin any multistatement transactions entered through SQLCI.

NonStop SQL/MP protects audited tables by requiring DML statements within programs to occur within explicitly defined transactions; however, NonStopSQL/MP also allows you to create nonaudited tables that are not protected even within transactions.

- Updates

ANSI/ISO SQL allows a positioned UPDATE on any updateable cursor. NonStop SQL/MP requires a FOR UPDATE clause on the UPDATE WHERE CURRENT statement for a positioned UPDATE.

- Other features

NonStop SQL/MP does not include these features:

- Delimited identifiers
- Explicit naming of select list elements
- Module language
- SQLSTATE status codes
- Updateable primary keys
- DECIMAL columns declared without explicit lengths
- Language embeddings for Ada, FORTRAN, and PL/1

SQL/MP Features From Intermediate Level SQL 1992

NonStop SQL/MP includes these features from Intermediate Level ANSI/ISO SQL, although the specific implementation of the features does not always conform exactly to the implementation described in the ANSI/ISO standard:

- DATETIME, NATIONAL CHARACTER, and VARCHAR data types
- The ability to add and drop constraints

(ANSI/ISO SQL provides ALTER TABLE ADD CONSTRAINT and ALTER TABLE DROP CONSTRAINT statements. NonStop SQL/MP provides a similar capability with the CREATE CONSTRAINT and DROP CONSTRAINT statements.)

- The CURRENT_TIMESTAMP function and the functionality of the CURRENT_DATE, CURRENT_TIME, and EXTRACT functions

(NonStop SQL/MP includes a CURRENT function and date-time field specifications in expressions that provide functionality equivalent to CURRENT_DATE, CURRENT_TIME, and EXTRACT.)

- The CAST function for parameters
- The ALTER TABLE ADD COLUMN statement
- The DROP TABLE and DROP VIEW statements
- Case-insensitive identifiers
- INNER JOIN and LEFT OUTER JOIN operators

(NonStop SQL/MP uses the keywords LEFT JOIN instead of LEFT OUTER JOIN for this operator.)

- Multiple built-in character sets (partial support only)
- Dynamic SQL, including:
 - Use of host variables for statement and cursor names
 - PREPARE, EXECUTE, and EXECUTE IMMEDIATE statements
 - DESCRIBE and DESCRIBE INPUT statements
 - USING DESCRIPTOR and USING clauses on OPEN CURSOR
 - RETURNING clause on the EXECUTE and INSERT
- Subqueries in comparison predicates that have GROUP BY and HAVING clauses or FROM clause references to grouped views
- UNION and UNION ALL in view definitions and subqueries
- ANSI/ISO SQL allows you to set the Isolation Level for a transaction to READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE. The default is SERIALIZABLE.

NonStop SQL/MP has access options that are similar to ANSI/ISO Isolation Levels. You can set the access option for an individual DML statement (but not for an entire transaction) to BROWSE, STABLE, or REPEATABLE. The default, STABLE, is equivalent to the ANSI/ISO Isolation Level READ COMMITTED.

Note that what SQL/MP calls `access options` are not related to the ANSI/ISO Access Mode for a transaction. NonStop SQL/MP has no equivalent for Access Mode.

SQL/MP Features From Full Level SQL 1992

NonStop SQL/MP includes these features from Full Level ANSI/ISO SQL, although the specific implementation of the features does not always conform exactly to the implementation described in the ANSI/ISO standard:

- DATETIME extensions (arbitrary precision for seconds)
- CREATE COLLATION and DROP COLLATION statements
- The ability to add and drop single-table assertions

(ANSI/ISO SQL provides CREATE ASSERTION and DROP ASSERTION statements. NonStop SQL/MP provides a similar capability for single-table assertions with CREATE CONSTRAINT and DROP CONSTRAINT statements.)

SQL/MP Extensions to SQL 1992

NonStop SQL/MP provides these extensions to ANSI/ISO SQL:

- Additional data types:
 - LARGEINT (64-bit integers) and PIC data types
 - UNSIGNED/SIGNED option for numeric data types
- Additional table features:
 - Ascending or descending order for user-defined primary keys
 - System-generated primary keys
 - Clustering keys
 - Distributed tables
 - System default values for columns
 - Key-sequenced, entry-sequenced, and relative file types
 - Partitioned tables
 - Nonaudited tables
 - Other Guardian file attributes for tables
- Additional DDL statements:

- CREATE INDEX
- ALTER {TABLE | INDEX | COLLATION | PROGRAM | VIEW}
- DROP {INDEX | PROGRAM}
- COMMENT ON
- HELP TEXT
- UPDATE STATISTICS
- Additional DCL statements and data control directives:
 - LOCK TABLE
 - UNLOCK TABLE
 - FREE RESOURCES
 - CONTROL TABLE
 - CONTROL EXECUTOR
 - CONTROL QUERY
- Additional functions:
 - CONVERTTIMESTAMP
 - DATEFORMAT
 - DAYOFWEEK
 - EXTEND
 - JULIANTIMESTAMP
 - SETSCALE
 - UPSHIFT
- Predicates can have multiple values (for example, $a, b < 10, 20$)
- SHARE and EXCLUSIVE lock modes let you share or restrict access to locked data.
- Objects referred to in SQL statements can be reassigned to different objects at run time by using DEFINES.
- The WHENEVER directive includes an SQLWARNING option and allows a CALL statement.

Statements

SQL/MP statements define SQL objects and catalogs, manipulate data within those objects and catalogs, and control various aspects of the processes that perform the data definition and manipulation you specify in the statements.

A single SQL/MP statement can contain up to 32,767 single-byte characters, including spaces. A tab is considered the same as a space in SQL/MP statements.

You can enter and execute most SQL/MP statements either from SQLCI or from an embedded SQL program. (SQLCI commands, in contrast to SQL/MP statements, can be entered only from SQLCI.)

Each SQL statement must end with a terminator that is not shown in the statement description because it depends upon the host language. The terminator for SQL statements in SQLCI, C, Pascal, and TAL is a semicolon (;). The terminator in COBOL is END EXEC.

For more information about entering SQL statements in SQLCI, see [SQLCI](#) on page S-60 and [SQLCI Commands](#) on page S-63. For general information about embedded SQL programs, see [Embedded SQL](#) on page E-2 or, for more information, see the SQL/MP programming manual for a specific language.

[Table S-4](#) on page S-77 summarizes the SQL/MP statements and directives. (A directive is a statement that gives instructions to a compiler.) Statements that can be used only in programs are indicated with an asterisk (*) following the statement name. For more information about a specific statement listed in the summary, see the entry for that statement.

Table S-3. Summary of SQL Statements

Command	Action
ALTER CATALOG Statement	Alters security for a catalog
ALTER COLLATION Statement	Renames or alters security for a collation
ALTER INDEX Statement	Renames, adds, or drops partitions, or alters security or other attributes of an index
ALTER PROGRAM Statement	Renames or alters security of an SQL program in a Guardian file
ALTER TABLE Statement	Renames, alters security or file attributes, or enables or disables similarity checks for a table. Also adds columns to a table and adds, drops, or moves partitions of a table.
ALTER CATALOG Statement	Alters security for a catalog
ALTER COLLATION Statement	Renames or alters security for a collation
ALTER INDEX Statement	Renames, adds, or drops partitions, or alters security or other attributes of an index

* Can be used only in program

Table S-3. Summary of SQL Statements

Command	Action
<u>ALTER PROGRAM Statement</u>	Renames or alters security of an SQL program in a Guardian file
<u>ALTER TABLE Statement</u>	Renames, alters security or file attributes, or enables or disables similarity checks for a table. Also adds columns to a table and adds, drops, or moves partitions of a table.
<u>ALTER VIEW Statement</u>	Renames, alters security, or enables or disables similarity checks for a view. Also adds columns to a view.
<u>BEGIN DECLARE SECTION Directive</u> *	Begins a section in a host program for declaring variables known to SQL
<u>BEGIN WORK Statement</u>	Starts a TMF transaction
<u>CLOSE Statement</u> *	Closes a cursor
<u>COMMENT Statement</u>	Writes a comment about an SQL object to a catalog
<u>COMMIT WORK Statement</u>	Commits all database changes made during the current TMF transaction and frees resources
<u>CONTINUE Statement</u>	Directs SQL to commit or cancel a DDL operation ready for its final phase
<u>CONTROL EXECUTOR Directive</u>	Enables or disables parallel processing of queries
<u>CONTROL QUERY Directive</u>	Specifies details of plans for queries, including when names are resolved, whether hash joins are permitted, and whether to optimize response time for returning few or many rows
<u>CONTROL TABLE Directive</u>	Specifies performance-related options for accesses to a table or view, including access path, join method, and many other options
<u>CREATE CATALOG Statement</u>	Creates a catalog
<u>CREATE COLLATION Statement</u>	Creates a collation
<u>CREATE CONSTRAINT Statement</u>	Creates a constraint for a table
<u>CREATE INDEX Statement</u>	Creates an index on a base table
<u>CREATE TABLE Statement</u>	Creates a base table
<u>CREATE VIEW Statement</u>	Creates a view
<u>DECLARE CURSOR Statement</u> *	Defines a cursor
<u>DELETE Statement</u>	Deletes rows from a table or view
<u>DESCRIBE Statement</u> *	Returns information about output variables of prepared statements

* Can be used only in program

Table S-3. Summary of SQL Statements

Command	Action
<u>DESCRIBE INPUT Statement</u> *	Returns information about input variables of prepared statements
<u>DROP Statement</u>	Drops a catalog, collation, constraint, index, table, view, or SQL program in a Guardian file
<u>END DECLARE SECTION Directive</u> *	Ends a section in a host program for declaring variables known to SQL
<u>EXECUTE Statement</u>	Executes a compiled statement
<u>EXECUTE IMMEDIATE Statement</u> *	Executes an SQL statement contained in a host variable
<u>FETCH Statement</u> *	Retrieves a row from a cursor
<u>FREE RESOURCES Statement</u>	Closes cursors and releases locks held by a process
<u>GET CATALOG OF SYSTEM Statement</u>	Retrieves the name of a local or remote system catalog
<u>GET VERSION Statement</u>	Retrieves the version of a specific SQL object, catalog, or system
<u>GET VERSION OF PROGRAM Statement</u>	Retrieves the PCV, PFV, or HOSV of an SQL program
<u>HELP TEXT Statement</u>	Specifies help text for a column of a table or view
<u>INCLUDE SQLCA Directive</u> *	Declares an area in a host program to receive run-time status information
<u>INCLUDE SQLDA Directive</u> *	Declares an area in a host program to receive information about input and output variables for dynamic SQL statements
<u>INCLUDE SQLSA Directive</u> *	Declares an area in a host program to receive execution statistics for DML statements
<u>INCLUDE STRUCTURES Directive</u> *	Specifies the structure version for INCLUDE SQLCA, INCLUDE SQLSA, and INCLUDE SQLDA directives
<u>INSERT Statement</u>	Inserts a row into a table or view
<u>INVOKE Directive and Command</u>	Generates a record description of a table or view
<u>LOCK TABLE Statement</u>	Locks a table (or the underlying tables of a view) and associated indexes
<u>OPEN Statement</u> *	Opens a cursor
<u>PREPARE Statement</u>	Compiles a DDL, DML, DCL, or DSL statement for later execution by EXECUTE
<u>RELEASE Statement</u> *	Deallocates memory for a dynamic SQL statement referred to through a host variable

* Can be used only in program

Table S-3. Summary of SQL Statements

Command	Action
<u>ROLLBACK WORK Statement</u>	Undoes all database modifications made to audited objects during the current TMF transaction and releases all locks held by the transaction
<u>SELECT Statement</u>	Retrieves data from tables and views
<u>SQL Directive</u>	Indicates to a host language compiler that a program contains SQL statements
<u>UNLOCK TABLE Statement</u>	Releases locks held on nonaudited tables or views
<u>UPDATE Statement</u>	Updates values in columns of a table or view
<u>UPDATE STATISTICS Statement</u>	Updates statistics about the contents of a table and its indexes
<u>WHENEVER DIRECTIVE</u> *	Specifies action to take when errors, warnings, or no-row-found conditions occur in a program

* Can be used only in program

Static SQL

Table S-4. Summary of SQL Statements

Command	Action
* Can be used only in program	

Static SQL is a form of embedded SQL in which SQL statements are coded directly into a host language source program.

Static SQL statements begin with EXEC SQL and end with one of these statement terminators:

- ;
END-EXEC. Occurs in C, Pascal, and TAL programs
- Occurs in COBOL programs

Unlike dynamic SQL statements, which can be constructed during program execution, static SQL statements must be coded directly in the host language program before compilation.

You can code static SQL statements to refer to different data bases during different executions by specifying table and view names with DEFINES and using CONTROL QUERY to request execution-time name resolution, but the SQL statements themselves remain unchanged (“static”) from execution to execution.

For more information about using static SQL, see the SQL/MP programming manual for one of the host languages.

This is an example of a static SQL statement from a COBOL program:

```
EXEC SQL
  SELECT LAST_NAME, EMPNUM INTO :LNAME, :EMPNUM
  FROM EMPLOYEE WHERE DEPTNUM = 200
END-EXEC.
```

This is an example of a static SQL statement from a C, Pascal, or TAL program:

```
EXEC SQL
  SELECT LAST_NAME, EMPNUM INTO :LNAME, :EMPNUM
  FROM EMPLOYEE WHERE DEPTNUM = 200;
```

Statistics

SQL has an UPDATE STATISTICS statement you can use to collect and save statistics on columns and tables. The SQL compiler uses these statistics to determine the selectivity of predicates, indexes, and tables. Because selectivity directly influences the cost of access plans and regular collection of statistics, it increases the likelihood that SQL will choose efficient access plans.

A NonStop SQL/MP installation should follow these rules for updating statistics:

- Do not use the UPDATE STATISTICS statement until you use FILEINFO with the STATISTICS option to see if performance problems are caused by fragmentation. By using FILEINFO, you should be able to determine if the performance is being impeded by fragmentation of blocks in a table. In that case, running UPDATE STATISTICS and recompiling the queries does not help. You should first reload the table online, by using the FUP RELOAD command.
- Determine the effect of UPDATE STATISTICS on production queries first. In an SQLCI session, issue a BEGIN WORK command and then issue UPDATE STATISTICS. Use EXPLAIN to see if the new statistics would give you the correct query plan. The UPDATE STATISTICS and EXPLAIN commands should be issued within a transaction so that the UPDATE STATISTICS operation can be backed out easily if necessary.
- Specify the NORECOMPILE option in the UPDATE STATISTICS statement so that dependent programs are not invalidated. By default, an UPDATE STATISTICS operation invalidates dependent programs. Even if UPDATE STATISTICS is executed within a transaction that can be backed out, if the NORECOMPILE option is omitted, dependent programs are still invalidated because program file labels are not audited. Updates to program file labels are not backed out.

Storage Management Foundation (SMF)

The SMF subsystem is an optional HP product. SMF simplifies name, storage, and file management with logical files, virtual volumes, and storage pools. On any SMF node, logical files, virtual volumes, and storage pools can coexist with conventional Guardian files and volumes. In the context of SMF, conventional Guardian files and volumes are called `direct files` and `direct volumes`.

With SMF, SQL objects can be either direct or logical files. A logical file is a file that resides on a virtual volume; a virtual volume is actually a process that uses a pool of physical volumes for storage. When you create an SQL object on a virtual volume, the object is a logical file, and you always refer to it by logical name. The object can be located on any physical volume in the storage pool for the virtual volume you specify; SMF automatically determines the best location.

Considerations—SMF

- Most SQL CREATE and ALTER statements have either syntax changes or considerations for SMF. Syntax changes consist of a new option, PHYSVOL, which allows you to place an SQL object on a particular physical volume associated with the specified virtual volume. Some SQL/MP utility commands are also affected, as are temporary table and file placement.

See also:

ALTER CATALOG Statement	DROP SYSTEM CATALOG Command
ALTER COLLATION Statement	DUP Command
ALTER INDEX Statement	FILEINFO Command
ALTER PROGRAM Statement	FILENAMES Command
ALTER TABLE Statement	GOAWAY Command
ALTER VIEW Statement	INVOKE Directive and Command
CLEANUP Command	LOAD Command
CONVERT Command	Parallel Index Loading
COPY Command	PARTITION Clause
CREATE CATALOG Statement	PURGE Command
CREATE COLLATION Statement	PURGEDATA Command
CREATE CONSTRAINT Statement	Qualified Fileset List
CREATE INDEX Statement	SECURE Command
CREATE SYSTEM CATALOG Command	Temporary Tables
CREATE TABLE Statement	UPGRADE CATALOG Command
CREATE VIEW Statement	VERIFY Command
DISPLAY USE OF Command	= _SORT_DEFAULTS DEFINE
DOWNGRADE CATALOG Command	= _SQL_EXE_USE_SWAPVOL DEFINE
DROP Statement	= _SQL_TM_node_vol DEFINE

- On physical volumes, subvolumes named ZYS* are reserved for SMF. Do not attempt to create objects or files on subvolumes named in this format.
- If you specify a logical file name in a FILEINFO, DETAIL command, the corresponding physical file name is displayed. Similarly, if you execute FILEINFO, DETAIL for a physical file, its corresponding logical name is displayed.

String Functions

You can use string functions in expressions that involve columns defined with character data types. You can use a string function anywhere an arithmetic expression is allowed.

NonStop SQL/MP provides these string functions:

SUBSTRING	Extracts a substring from a given string.
POSITION	Searches for a given character pattern in a character string. If the pattern is found, SQL returns the position of the pattern.
OCTET_LENGTH	Returns the length of a character string in bytes.
CHAR_LENGTH	Returns the number of characters in a string.
TRIM	Removes leading or trailing characters from a character string.

For more information, see the descriptions of specific functions and [Character Expressions](#) on page C-14.

String Literals

A string literal represents a series of characters and consists of that series of characters surrounded by double or single quotation marks, optionally preceded by a clause that specifies the character set associated with the characters.

```
[ _ISO88591 ]
[ _ISO88592 ]
[ _ISO88593 ]
[ _ISO88594 ]
[ _ISO88595 ]
[ _ISO88596 ]
[ _ISO88597 ] { 'string' }
[ _ISO88598 ] { "string" }
[ _ISO88599 ]
[ _KANJI ]
[ _KSC5601 ]
[ _UNKNOWN ]
[ N ]
```

`_ISO88591 ... _KSC5601`

associates a character set with the string literal.

The character set is one of the single-byte character sets ISO 8859/1 through ISO 8859/9 or one of the double-byte character sets Kanji or KSC5601. For more information about character sets, see [Character Sets](#) on page C-17.

`_UNKNOWN`

specifies that data in the string literal belongs to an unknown character set. Specifying UNKNOWN is equivalent to omitting the character set specification. NonStop SQL/MP handles the data as if it were 8-bit data.

`N`

associates the system default multibyte character set with the string literal. (Kanji is the standard system default multibyte character set, but the default can be different at your site. For more information, see [Multibyte Character Sets](#) on page M-44)

string

is a series of single-byte or double-byte characters.

Considerations—String Literals

- As indicated in the syntax diagram, you must enclose string literals in either single (') or double (") quotation marks, although the value of the string literal does not include the quotation marks. To specify the delimiter character within the literal, use two consecutive quotation marks.
- A string literal can be as long as a character column. For limits on data length, see [Limits](#) on page L-6.
- To specify a long string literal in SQLCI, separate the literal into several smaller string literals and use line-continuation characters (&) to connect them.

In host programs, rules for breaking string literals across lines depend on the language being used.

- In string comparisons, lowercase letters are not equivalent to the corresponding uppercase letters.
- Strings associated with double-byte character sets must contain an even number of bytes. An error occurs if such a string contains an odd number of bytes.
- You should not mix single-byte and double-byte characters in the same string, but NonStop SQL/MP does not prevent you from doing so.
- NonStop SQL/MP does not perform data validation to ensure that characters in a string literal belong to the character set associated with the string literal. For example, this string literal is valid:

```
_KANJI "abcdef "
```

Examples—String Literals

- This string literals do not have an associated character set:

```
"This is an ordinary literal."
"This literal contains "" a quotation mark."
"This SQLCI literal is" &
" in three parts, " &
"specified over three lines"
'1234.56'
_UNKNOWN"abc^&*"

```

- This string literals have associated character sets. The first is associated with a single-byte character set, and the second and third are associated with double-byte character sets. The third is associated with either KANJI or KSC5601, depending on the default national character data type in effect on the node.

```
_ISO88594'SMITH, JOHN'
_KANJI"c1c2c3c4c5c6"
N"c1c2c3c4c5c6c7c8c9"

```

Subqueries

A subquery is a special form of the **SELECT** statement that selects only for comparison. You can specify a subquery in a comparison, **EXISTS**, **IN**, or qualified predicate of a search condition.

```
( SELECT [ ALL | DISTINCT ] { expression
                             *
                             correlation-name.*
                             table-name.*
                             view-name.*
                           }

FROM table-ref [, table-ref] ...

[ WHERE search-condition]

[ HAVING search-condition]

[ [ FOR ] { BROWSE | STABLE | REPEATABLE } ACCESS ]

[ IN { SHARE | EXCLUSIVE } MODE ]

[ GROUP BY column-name [, column-name] ] ...

[ UNION [ ALL ] select-statement] )

```

For the complete syntax and rules for the elements and clauses of a subquery, see [SELECT Statement](#) on page S-18. For examples that use subqueries, see [Comparison Predicate](#) on page C-58 or [EXISTS Predicate](#) on page E-12.

Considerations—Subqueries

- A SELECT statement that contains a subquery is called an *outer query*. The subquery within the SELECT is called an *inner query*. The differences between a SELECT statement and a subquery are:
 - A subquery is always enclosed in parentheses.
 - A subquery returns a result table of one column.
 - A SELECT statement can retrieve values to place in a cursor or host variable. A subquery searches for values to use in comparisons. The comparisons determine whether a search condition is satisfied.
 - The INTO clause of a SELECT statement cannot be associated with a subquery in a cursor declaration. Therefore, the subquery cannot be used to retrieve values for host variables.
 - A SELECT statement can specify a list of elements to select. A subquery can specify only a single column or expression. You can, however, specify an asterisk or a correlation name followed by an asterisk if the subquery occurs in an EXISTS predicate or if the FROM clause refers to a single table (or view) consisting of a single column.
 - A subquery cannot contain an ORDER BY clause.
 - If a subquery is not part of an EXISTS, IN, or quantified predicate, and the subquery evaluates to more than one row, a run-time error occurs.
 - If a subquery contains references to an outer query, the subquery might be evaluated repeatedly. This type of subquery is called a *correlated subquery*, discussed later in this subsection.
- An outer query (a main SELECT statement) can have up to 15 levels of nested subqueries.

Subqueries within the same ON, WHERE, or HAVING clause are at the same level. For example, this query has one level of nesting:

```
SELECT * FROM TABLE1
  WHERE A = (SELECT P FROM TABLE2 WHERE Q = 1)
        AND B = (SELECT X FROM TABLE3 WHERE Y = 2)
```

A subquery within the WHERE clause of another subquery defines a new level, so this query has two levels of nesting:

```
SELECT * FROM TABLE1
  WHERE A = (SELECT P FROM TABLE2
            WHERE Q = (SELECT X FROM TABLE3
                    WHERE Y = 2))
```

- In the search condition of a subquery, you can refer to columns of any table or view defined in an outer query. Such a reference is called an *outer reference*. A subquery containing an outer reference is called a *correlated subquery*.

Each time an outer query selects and evaluates a row, the outer reference is visible as a new value to the correlated subquery. The correlated subquery operates on the new value to test its own search condition, so the correlated subquery executes whenever the outer query selects a new row, which leads to reduced performance.

If you refer to a column name that occurs in more than one outer query, you must qualify the column name with the correlation name of the table or view to which it belongs. The correlation name is known to other subqueries at the same level or to inner queries but not to outer queries.

If you use the same correlation name at different levels of nesting, an inner query uses the name from the nearest outer level. SQL checks the FROM clause of the subquery first, then its outer query, and so forth, until it determines the applicable table or view.

SUBSTRING Function

The SUBSTRING function extracts a substring out of a given string.

```
{ SUBSTRING ( character-string FROM start-position      }
{                               [ FOR substring-len ] ) }
```

where *character-string* is:

```
{ string-literal           }
{ column-name             }
{ param-name              }
{ host-var-name           }
{ UPSHIFT function         }
{ character-expression    }
```

start-position and *substring-length* are:

```
{ numeric-literal         }
{ column-name             }
{ param-name              }
{ host-var-name           }
{ expression              }
```

character-string

specifies the source string from which to extract the substring.

start-position

specifies the starting position within *character-string*, in number of characters, at which to start extracting the substring. *start-position* must be a value with an exact numeric data type and a scale of 0; otherwise, SQL returns an error.

substring-length

specifies the number of characters to extract from *character-string*.

substring-length must be a value of exact numeric data type with a scale of zero; otherwise, SQL returns an error. *substring-length* must be greater than or equal to zero.

Considerations—SUBSTRING Function

- The data types of *substring-length* and *start-position* must be numeric; otherwise, SQL returns an error.
- If the sum of *start-position* and *substring-length* is greater than the length of the character string, SQL returns the substring from *start-position* to the end of the string.
- If the sum of *start-position* and *substring-length* is less than zero, SQL returns an empty string ("").
- If *start-position* is greater than the length of the character string, SQL returns an empty string ("").
- If you do not specify *substring-length*, SQL returns all characters starting at *start-position* and continuing until the end of *character-string*.
- The resulting substring is always of type VARCHAR, with the same collating sequence and character set as the source character string. If the source character string is an upshifted CHAR or VARCHAR string, the result is an upshifted VARCHAR type.
- The resulting collating sequence and character set are the same as that of the string operand, *character-string*.
- If *character-string*, *start-position*, or *substring-length* is a null value, the result is null.

Examples—SUBSTRING Function

- This example returns "John":

```
SUBSTRING("Robert John Smith" FROM 8 FOR 4)
```
- This example returns "John Smith":

```
SUBSTRING("Robert John Smith" FROM 8)
```
- This example returns "Robert John Smith":

```
SUBSTRING("Robert John Smith" FROM 1 FOR 17)
```
- This example returns "John Smith":

```
SUBSTRING ("Robert John Smith" FROM 8 FOR 15)
```

- This example returns “”:

```
SUBSTRING ("Robert John Smith" FROM -5 FOR 2)
```

- This example returns “Ro”:

```
SUBSTRING ("Robert John Smith" FROM -2 FOR 5)
```

- This example produces an empty string, “”, which is different than a null value for the result:

```
SUBSTRING("Robert John Smith" FROM 8 FOR 0)
```

SUBTOTAL Command

SUBTOTAL is an SQLCI report writer command that specifies columns to subtotal and when to print the subtotals. SUBTOTAL returns you to the first SELECT output row.

```
SUBTOTAL column [ , column ] ... [ OVER break-col ] ;
```

column

identifies a column to subtotal. *column* must be a column with a numeric data type from the current detail line that is not an IF/THEN/ELSE column. You can specify it as a column name, an alias, a detail alias, or COL *number* (which specifies the position of the column in the select list).

OVER *break-col*

specifies that the subtotal prints when the data value of the named break column changes. *break-col* must be a break column specified in a BREAK ON command, but you can specify it as a column name, an alias, a detail alias, or COL *number* (which specifies the position of the column in the select list).

If you omit the OVER clause, subtotals print when the value in any currently defined break column changes.

Considerations—SUBTOTAL

- You must enter a BREAK ON command to define break columns you specify in a SUBTOTAL command. The BREAK ON command does not have to precede the SUBTOTAL command, but you must enter it before you list the report.
- When you enter a SUBTOTAL command for a specific break column, it replaces any existing SUBTOTAL command for that break column. If you enter a SUBTOTAL command without the OVER clause, it replaces any previous SUBTOTAL command without an OVER clause.
- A subtotal prints immediately beneath the print item it subtotals, and the subtotal value prints in the same display format as the print item. (When you specify a print item in a DETAIL command, check that the display format in the AS clause allows

enough room for the subtotal. If the subtotal is too large for its display format, the field is filled instead with overflow characters.)

Subtotals print on three lines: the first line contains underline characters (see [UNDERLINE_CHAR Option](#) on page U-1), the second line contains the subtotal value, and the third line is blank.

To identify the break group for the subtotal, the report writer prints a subtotal label (see [SUBTOTAL_LABEL Option](#) on page S-88) under the break column. If the label does not fit in the break column, the label is truncated. An asterisk is the default subtotal label.

If the subtotal column is the same as the break column, both the subtotal label and the subtotal value must print under that column. If the column is wide enough to accommodate both the label and the value, both are printed; otherwise, the label is truncated and can be entirely overwritten by the subtotal value.

- In calculating subtotals, the report writer uses the maximum format for the item's data type and the same scale as the item to be subtotaled. In unusual cases (such as when an expression contains an item multiplied by an extremely small fractional value) this strategy can cause numeric overflow.

Specifying small numeric values in exponential notation (for example, .0000246615 E0 instead of .0000246615) can prevent overflow by causing the report writer to use a floating-point format for such calculations.

Examples—SUBTOTAL

- This example selects data ordered by the value in column ORDERNUM, then generates a subtotal for the expression used as column 3 whenever the value of ORDERNUM changes:

```
>> SET LIST_COUNT 0;
>> SELECT ORDERNUM, PARTNUM, (QTY_ORDERED * UNIT_PRICE)
+>   FROM SALES.ODETAIL ORDER BY ORDERNUM;
S> BREAK ON ORDERNUM;
S> SUBTOTAL COL 3 OVER ORDERNUM;
```

- This example selects data ordered by the values in columns DEPTNUM and JOBCODE, and then generates subtotals for salaries and bonuses for each job code. (The example uses an asterisk as the default subtotal label. To learn how to change the default subtotal label, see [SUBTOTAL_LABEL Option](#) on page S-88.)

```
>> SET LIST_COUNT 0;
>> SELECT DEPTNUM, JOBCODE, SALARY, SALARY*.025
+>   FROM PERSNL.EMPLOYEE ORDER BY DEPTNUM, JOBCODE;
S> DETAIL DEPTNUM, JOBCODE, SALARY AS F15.2, COL 4 AS F12.2
+>   HEADING "BONUS";
S> BREAK ON DEPTNUM, JOBCODE;
S> SUBTOTAL SALARY, COL 4;
S> LIST ALL;
```

The report looks like this:

DEPTNUM	JOBCODE	SALARY	BONUS
1000	100	137000.10	3425.00
	*	137000.10	3425.00
	500	25000.75	625.02
		29000.00	725.00
		50000.00	1250.00
	*	104000.75	2600.02
*		260000.85	6500.02
1500	100	90000.00	2250.00
	*	90000.00	2250.00
	600	26000.00	650.00
		32000.00	800.00
	*	58000.00	1450.00
	900	17000.00	425.00
	*	17000.00	425.00
*		165000.00	4125.00

SUBTOTAL_LABEL Option

SUBTOTAL_LABEL is an option of the SQLCI report writer command SET STYLE. SUBTOTAL_LABEL specifies a label to print in a break column on the same line as a subtotal.

```
SUBTOTAL_LABEL "label"
```

label

is a character string to use as a label. *label* consists of 0 to 255 single-byte characters or 0 to 127 double-byte characters. The default is * (asterisk).

Considerations—SUBTOTAL_LABEL

- The report writer uses the same subtotal label for all groups. The label prints in the break column associated with the subtotal and is truncated if it does not fit. For more information and for a sample report with subtotal labels, see [SUBTOTAL Command](#) on page S-86.
- If you do not want subtotal labels to print, specify an empty string ("") in the SUBTOTAL_LABEL command.

Example—SUBTOTAL_LABEL

This command defines a subtotal label:

```
>> SET STYLE SUBTOTAL_LABEL "@" ;
```

SUM Function

SUM is a function that computes the sum of a set of numbers.

The type of the result depends on the type of the argument. If the argument is an exact numeric type, the result is LARGEINT. If the argument type is FLOAT, REAL, or DOUBLE PRECISION, the result is DOUBLE PRECISION.

The scale of the result is the same as the scale of the argument. If the argument has no scale, the result is truncated.

<pre>SUM { ([ALL] <i>expression</i>) } { (DISTINCT <i>column</i>) }</pre>
--

`[ALL] expression`

specifies a numeric or INTERVAL expression that indicates the set of values to sum.

The expression must include a value from each row of the result table (that is, at least one column from the result table) and cannot include the COUNT, MAX, MIN, or AVG functions, or another SUM function, as shown:

```
SUM (SALARY)
SUM (PARTCOST * QTY_ORDERED)
```

ALL is an optional keyword that does not change the meaning of the clause. SQL uses all rows (whether or not you specify ALL) unless you use the DISTINCT clause, described next.

`DISTINCT column`

specifies a set of distinct column values to average from each row of the result table. The column cannot be a column from a view that corresponds to an expression in the view definition.

If you specify DISTINCT in more than one SUM function in the same statement, the functions must reference the same column.

Considerations—SUM

- SUM is evaluated after eliminating all null values from the aggregate set. If the result set is empty, SUM returns a null value.

- A host variable that receives the result of the SUM function must have an indicator variable to handle a possible null value. (For more information about using indicator variables, see the SQL/MP programming manual for your host language.)

Example—SUM

To compute the total value of parts in the current inventory (the sum of each value in the PRICE column multiplied by the corresponding value in the QTY_AVAILABLE column), type this:

```
>> SELECT SUM (PRICE * QTY_AVAILABLE) FROM SALES.PARTS;
(EXPR)
-----
83052750.00
--- 1 row(s) selected.
```

Super ID

The super ID is a Guardian user ID with group number 255 and user number 255. The super ID is intended for system maintenance and administration and has many special privileges.

The super ID can read, write to, execute, and purge any object on the local node. The super ID can also resecure and alter attributes of objects on the local node.

A super ID on one node in a network does not have super ID privileges on other nodes in the network. Most super ID privileges require being directly logged on the local node as the super ID. However, a super ID with a remote password (not a logon password) to another node has group manager privileges (generalized owner privileges) for objects and files secured so that their owner can purge them remotely.

Syskeys

A SYSKEY, or system-defined primary key, is a primary key defined by SQL rather than by the user.

Tables stored in relative and entry-sequenced files or in key-sequenced files without a user-defined primary key have a primary key defined by SQL and stored in a column named SYSKEY.

SQL adds the SYSKEY column to the table definition for you; you do not supply it. SYSKEY is the first column in the table, and its data type depends on the organization of the file, as shown:

File Organization	SYSKEY Data Type	SYSKEY Value
Entry-sequenced	INTEGER UNSIGNED	4-byte record address
Key-sequenced	LARGEINT SIGNED	8-byte unique number
Relative	INTEGER UNSIGNED	4-byte record number

When you insert a record in a table stored in an entry-sequenced file or in a key-sequenced file with a SYSKEY column, the file system automatically generates a value for the SYSKEY column. You cannot supply the value.

When you insert a record in a table stored in a relative file, you can specify a record number for the SYSKEY value. If you do not specify a value, the file system supplies a record number. The SYSKEY column cannot contain a null value.

Because a SYSKEY is defined by the system when a record is inserted, an SQLCI LOAD or SQLCI COPY of a table with a SYSKEY (or a clustering key, which uses a SYSKEY) changes the SYSKEY value for the records in the target table. Any previous SYSKEY values stored in other tables no longer point to the correct row after such an SQLCI LOAD or SQLCI COPY operation. Moving a partition does not change the SYSKEY values, although the MOVE operation includes a LOAD operation.

Note. A FUP RELOAD does not modify the SYSKEY value of the rows in a table.

You cannot update values in the SYSKEY column of any table, but you can use the SELECT statement to query SYSKEY values. If SYSKEY is provided in the value list or for a query, the value range allowed is 0 through 4,294,963,199 (for a 4-byte SYSKEY column).

The catalog description of a table with a SYSKEY reflects the presence of the SYSKEY column, but SQL does not return the value of the SYSKEY column unless a query explicitly selects that column. For example, this SELECT statement would not display SYSKEY values:

```
SELECT * FROM table-name
```

If a view definition explicitly includes the SYSKEY column of a table, however, a SELECT * on the view does return SYSKEY values.

You cannot partition key-sequenced tables that use SYSKEY as the primary key.

System Catalog

Each node on a network has one special catalog called the system catalog.

The system catalog contains the same tables as other catalogs on the node, but includes one additional table, the CATALOGS table, that lists all catalogs on the node. For more information, see [CATALOGS Table](#) on page C-11.

The system catalog is established during the installation of NonStop SQL/MP. By default, it is located on volume \$SYSTEM and subvolume SQL, although you can specify a different volume and subvolume at installation time if necessary. (The CATALOGS table is always located on subvolume SQL on the same volume as the system catalog.)

These statements can operate on the system catalog:

- ALTER CATALOG
- CREATE SYSTEM CATALOG
- DOWNGRADE SYSTEM CATALOG
- DROP SYSTEM CATALOG
- GET CATALOG OF SYSTEM (returns location)
- GET VERSION OF CATALOG
- UPGRADE SYSTEM CATALOG

However, the ALTER CATALOG statement does not affect the system catalog CATALOGS table. You must alter that table with the ALTER TABLE statement.

If you have appropriate authorization, you can also use other SQL statements to operate on individual tables within the system catalog. For more information, see [Catalogs](#) on page C-8.

SYSTEM Command

SYSTEM is an SQLCI session command that selects a node to be the current default node for the SQLCI session.

```
SYSTEM [ \node ] ;
```

node

is the name of a node to be the current default. If you omit *node*, SQL uses the node on which SQLCI is executing.

Considerations—SYSTEM

- The default stays in effect only for the SQLCI session. When you return to TACL, the TACL default is used.
- These commands are not equivalent:

```
>> SYSTEM \local-node;
>> SYSTEM ;
```

The first command shown causes the network restrictions on file-name lengths to take effect; the second command does not. For information on network file-name lengths, see the *Expand Network Management Guide*.

- SYSTEM sets the node name qualifier in the =_DEFAULTS DEFINE. (You can also change to a different default node by using the ALTER DEFINE =_DEFAULTS command and changing the node specified for the VOLUME attribute.)

Example—SYSTEM

This command makes \SYS1 the default node:

```
>> SYSTEM \SYS1;
```

System DEFINES

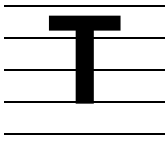
A system DEFINE is a DEFINE used in HP NonStop system software to identify system defaults or configuration information.

Each system DEFINE name begins with an equal sign and an underscore (= _). This special prefix is reserved for system DEFINE names. (Do not create DEFINE names that begin with an equal sign and an underscore unless specifically directed to do so in HP documentation.)

SQL uses these system DEFINES:

=_AUDSERV_XSWAP_ <i>node</i>	Specifies a swap volume for the audit fix-up process on <i>node</i>
=_DEFAULTS	Sets defaults for Guardian name expansion
=_SORT_DEFAULTS	Specifies defaults for FastSort operations
=_SQL_CMP_EVENT	Directs SQL to log SQL compiler event messages to a file or home terminal
=_SQL_CMP_EVENT_NO0	Suppresses logging of SQL compiler event messages to \$0
=_SQL_ <i>cmp_node</i>	Identifies files for components of the SQL/MP software
=_SQL_EXE_USE_SWAPVOL	Directs SQL to allocate temporary tables for serial plans on the swap volume for the process
=_SQL_MSG_ <i>node</i>	Sets file for SQL message text
=_SQL_RECGEN_ <i>node</i>	Specifies an alternate location for the FastSort record generator
=_SQL_TM_ <i>node_vol</i>	Directs SQL to use another disk for temporary tables normally allocated on the specified volume; optionally opens the tables with SYNCDEPTH 1 instead of 0

For more information about a specific system DEFINE, see the entry for that DEFINE.



TABLECODE File Attribute

TABLECODE is a file attribute that assigns a numeric code to a table, index, or collation. TABLECODE applies to key-sequenced, relative, and entry-sequenced files and to indexes.

A table code is a numeric code chosen by the user to categorize files. The table code appears in the TABLES and INDEXES catalog tables.

In output from the FILEINFO command and in contexts other than SQL, a table code is referred to as a file code.

TABLECODE <i>code</i>

code>

is an integer value from 0 to 99 or from 1000 to 65535 that specifies the file code.

Values from 100 to 999 are reserved for use by HP.

Values from 571 to 599 are reserved for use by NonStop SQL/MP.

The table default is TABLECODE 0.

The index default is its table's value at index creation.

Tables

A table is a logical representation of data in a database in which a set of records is represented as a sequence of rows, and the set of fields common to all records is represented by columns. The intersection of a row and column represents the data value of a particular field in a particular record.

All data in a NonStop SQL/MP database is stored in tables. Each table is described in an SQL/MP catalog and stored in a physical file in the Guardian environment of a system.

The description of a table includes the name of the table, the name of each column of the table, the type of data you can store in each column of the table, and other information about the table, including the physical characteristics of the file that stores the table.

In some descriptions of SQL, tables are referred to as base tables to distinguish them from views, which are logical tables.

This approach shifts the processing time for opening objects to the beginning of program execution. The processing time is not counted from the time when the individual statements that require the objects are processed. After an object is opened, it stays open until the program is stopped.

By default, SQL automatically opens partitions of tables and indexes as they are needed. You can, however, use the **CONTROL TABLE** statement with the **OPEN ALL** option to open all indexes and base partitions of a table the first time a partition is accessed.

Note. Use the **CONTROL TABLE OPEN ALL** statement with the **OPEN ALL** option only if all these are true:

- All open activities must occur when the program starts (add a dummy call to the cursor during initialization).
 - The object containing the cursor eventually accesses all partitions.
 - The plan for the cursor is not a parallel plan.
-

For more information about tables, see [CREATE TABLE Statement](#) on page C-154.

TABLES Table

The **TABLES** table is a catalog table that contains information about tables, views, and collations. It contains a row for each table, view, and collation in the catalog, including itself and other catalog tables.

[Table T-1](#) describes the contents of the **TABLES** table.

Table T-1. The TABLES Table (page 1 of 2)

Column Name	Data Type	Description
1 TABLENAME*	CHAR(34)	Name of table, view, or collation
2 TABLETYPE	CHAR(2)	TA if table VI if view CP if collation
3 TABLECODE	SMALLINT UNSIGNED	Code for type of table. Codes 100-999 indicate reserved for HP use. Other numbers are values of the TABLECODE file attribute
4 COLCOUNT	SMALLINT SIGNED	Number of columns in table or view
5 GROUPID	SMALLINT UNSIGNED	Group number of owner's user ID
6 USERID	SMALLINT UNSIGNED	User number of owner's user ID
7 CREATETIME	LARGEINT SIGNED	Julian timestamp from table creation
8 REDEFTIME	LARGEINT SIGNED	Julian timestamp from alteration that invalidated dependent programs (updated by UPDATE STATISTICS)
9 SECURITYVECTOR	CHAR(4)	Security string for table, view, or collation

Table T-1. The TABLES Table (page 2 of 2)

Column Name	Data Type	Description
10 SECURITYMODE	CHAR(1)	S if Safeguard security G if Guardian security
11 OBJECTVERSION	SMALLINT UNSIGNED	Version of table, view, or collation
12 SIMILARITYCHECK	VARCHAR(3 0)	ENABLED if similarity check is allowed, DISABLED if not

* Indicates primary key

The columns TABLENAME through SECURITYVECTOR (1 through 9) were created in version 1. The columns SECURITYMODE through OBJECTVERSION (10 through 11) were added in version 300. Column SIMILARITYCHECK (12) was added at version 310.

Guardian names in the TABLES table are fully qualified and use uppercase characters.

TEDIT Command

TEDIT is an SQLCI command that invokes the PS Text Edit editor.

```
TEDIT [tedit-command-line];
```

tedit-command-line

is one or more TEDIT commands, as described in the *PS Text Edit Reference Manual*.

If you specify more than one command, you must use a semicolon between commands and enclose *tedit-command-line* in quotation marks, as shown:

```
TEDIT "QUERY6; LASTPAGE";
```

Example—TEDIT

This example invokes TEDIT from SQLCI:

```
>> TEDIT;
```

Temporary Tables

SQL creates temporary tables as needed according to these rules:

- If a query involves a join operation, SQL creates a temporary table on the volume on which the outermost table in the join resides. If the table is partitioned, SQL creates a temporary table on the volume on which the specified partition resides.
- If a query involves a hash join or a hash grouping, SQL creates temporary files on the program swap volume.
- If a query requires a sort operation but does not use FastSort, SQL creates any temporary tables used for the sort on the volume on which the table being sorted resides. If the table is partitioned, SQL creates any temporary table on the same partition as the partition being sorted.
- If SQL uses FastSort to sort a table, FastSort's SORTPROG process creates a temporary file on the SORTPROG scratch volume.

To redirect temporary tables created by SQL, use the `=_SQL_TM_node_vol` DEFINE.

To redirect temporary tables created by SORTPROG, use the `=_SORT_DEFAULTS` DEFINE.

If SMF is installed on your node, temporary table placement for serial plans is unchanged. For example, if the outermost table in a join resides on a virtual volume, SQL creates a temporary table on the same virtual volume. However, for parallel plans SQL places temporary tables only on physical volumes. Virtual volumes are never candidates for temporary tables SQL creates during a parallel operation.

TIME_FORMAT Option

TIME_FORMAT is an option of the SQLCI report writer SET STYLE command that defines a default print format for items specified with AS TIME *.

```
TIME_FORMAT "time-format"
```

time-format

is a format to use as the default format for print items specified with AS TIME *. For information about how to specify formats, see [AUDIT File Attribute](#) on page A-74.

The default format is HP2:M2:S2.

Example—TIME_FORMAT

This example sets a new default time format:

```
>> SET STYLE TIME_FORMAT "HB:M2:S2:C2:T3";
```

An example of this format is:

```
3:45:20:16:033
```

TIME Data Type

Data of type TIME represents a time of day according to a 24-hour clock. Values of data type TIME are equivalent to values of data type DATETIME declared:

```
DATETIME HOUR TO SECOND
```

For more information, see [DATETIME Data Type](#) on page D-15.

Example—TIME Data Type

These literals are of data type TIME in (respectively) default, USA, and European format:

```
TIME "13:40:05"
TIME "1:40:05 PM"
TIME "13.40.05"
```

For more information, see [Date-Time Literals](#) on page D-10.

TIMESTAMP Data Type

Data of type TIMESTAMP represents a date according to the Gregorian calendar and a time of day according to a 24-hour clock. Values of data type TIMESTAMP are equivalent to values of data type DATETIME declared as:

```
DATETIME YEAR TO FRACTION(6)
```

For more information, see [DATETIME Data Type](#) on page D-15.

Example—TIMESTAMP Data Type

These literals are of data type TIMESTAMP in (respectively) default, USA, and European format:

```
TIMESTAMP "1990-01-22:13:40:05.55"
TIMESTAMP "01/22/1990 01:40:05.55 PM"
TIMESTAMP "22.01.1990 01.13.05.55 PM"
```

For more information, see [Date-Time Literals](#) on page D-10.

TMF Transactions

The TMF subsystem simplifies the task of maintaining data consistency for a distributed database being updated by concurrent transactions.

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

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

If, however, the changes cannot be made or the user does not want to complete the transaction, the transaction can be aborted so the database is rolled back to its original state.

Transaction Control Statements

The statements in this table control TMF transactions.

Function	Language	Transaction Control Statement
Start	SQL/MP	BEGIN WORK
	COBOL	ENTER TAL BEGINTRANSACTION
	C, Pascal, TAL	BEGINTRANSACTION
	SCREEN COBOL	BEGIN-TRANSACTION
Commit	SQL/MP	COMMIT WORK
	COBOL	ENTER TAL ENDTRANSACTION
	C, Pascal, TAL	ENDTRANSACTION
	SCREEN COBOL	END-TRANSACTION
Abort	SQL/MP	ROLLBACK WORK
	COBOL	ENTER TAL ABORTTRANSACTION
	C, Pascal, TAL	ABORTTRANSACTION
	SCREEN COBOL	ABORT-TRANSACTION

Statements that control TMF transactions can be used from SQLCI or coded in an application.

For the TMF subsystem, the statements for each function are equivalent. Error processing, however, depends on the language of the statement.

User-Defined and System-Defined Transactions

TMF transactions you define are called *user-defined transactions*. In some cases, NonStop SQL/MP defines transactions for you. These transactions are called *system-defined transactions*. System-defined transactions and the operations that cause them are discussed in later sections.

To ensure that several statements either execute successfully or not at all, you can define one transaction consisting of several statements by using the **BEGIN WORK** and **COMMIT WORK** statements. You can abort a transaction with the **ROLLBACK WORK** statement.

COMMIT WORK and **ROLLBACK WORK** perform the **FREE RESOURCES** operation along with transaction control. **COMMIT WORK** is equivalent to this sequence:

FREE RESOURCES (SQL statement)

ENDTRANSACTION (procedure call)

ROLLBACK WORK is equivalent to:

FREE RESOURCES (SQL statement)

ABORTTRANSACTION (procedure call)

Rules for DDL and DML Statements

These rules apply to DDL statements in transactions:

- A DDL statement always executes within a TMF transaction or as a series of system-defined TMF transactions.
- Some DDL statements on audited tables can execute within a user-defined transaction. Other DDL statements on audited tables, and all DDL statements on nonaudited tables, cannot execute within a user-defined transaction.

DDL statements that cannot execute in a user-defined transaction automatically execute in a system-defined transaction or a series of system-defined transactions. An error occurs if you include these statements in user-defined transactions.

Whether a DDL statement on an audited table can execute within a user-defined transaction depends on the statement and on the options you specify on the statement. If a statement cannot be used within a user-defined transaction, the entry for that statement says so.

These rules apply to DML statements in transactions:

- DML statements in audited tables, views of audited tables, and mixed views must be performed within a TMF transaction except when reading data with **BROWSE ACCESS**.
- If deadlock occurs, the statement receives error 73 and is canceled; the transaction continues.

Note. The TMF subsystem works only on audited objects, so a transaction does not protect operations on nonaudited objects. The simplest approach is to make all tables audited. Audited is the default.

Rules for SQLCI

In SQLCI, you do not have to define your own TMF transactions or directly secure locks on data. By setting the AUTOWORK session option, you can specify system-defined transactions or user-defined transactions.

- **AUTOWORK ON**

If you set AUTOWORK to ON, all data in SQL operations is automatically accessed within a TMF transaction. By default, SQL obtains locks to guarantee a specific level of data consistency. You can control the level of data consistency by overriding the defaults provided by the locking mechanism.

SQL defines a transaction or a series of transactions for each DDL statement and for each DML statement. A TMF transaction does not affect DML operations on nonaudited objects, however.

The COPY, PURGE, and SECURE utilities also execute within a TMF transaction or series of transactions when they operate on audited SQL objects.

- **AUTOWORK OFF**

If you set AUTOWORK to OFF, you can define your own TMF transaction for statements that can execute within a user-defined transaction. You must define a transaction for DML operations on audited tables.

SQLCI automatically defines a transaction or series of transactions for a DDL statement unless a user-defined transaction is in effect.

Rules for Host Programs

- In a host program, you do not have to define a TMF transaction for a DDL statement, but if the statement can execute within a user-defined transaction, you can do so if necessary. (SQL automatically starts a system-defined transaction or series of transactions for a DDL statement on an audited table that occurs outside a user-defined transaction, ending the final transaction for the statement when the DDL statement finishes.)

In a host program, however, you must define a TMF transaction for a DML statement that operates on an audited object.

- Typically, only one TMF transaction at a time is in progress for an application.
- Either the requester or the server can start the transaction. Any program in a program unit can start the transaction.
- A context-free server should close its cursors and free locks on nonaudited tables before replying to the requester.
- If a host program executes a DDL statement on audited objects within a user-defined TMF transaction, the DDL operation is performed as part of the user-defined TMF transaction. If the operation terminates successfully, the host program determines whether to commit or abort the transaction. In contrast, if the

DDL operation terminates with an error and the operation was partially performed, the system automatically aborts the user-defined TMF transaction.

- DDL statements that operate on nonaudited tables cannot execute when a user-defined TMF transaction is in progress.
- A DML statement access option should provide transaction consistency and concurrency appropriate for your application and the environment in which it runs.
- When your program updates both audited and nonaudited tables in a TMF transaction, remember that only the audited tables are protected against inconsistency by TMF.

For example, suppose that a transaction updates two audited tables, A and C, and an unaudited table, B. Within the transaction, your program updates tables A and B, detects an error when attempting to update table C, and aborts the transaction. TMF undoes the changes to table A, but the changes to table B remain, making the database inconsistent.

TOTAL Command

TOTAL is an SQLCI report writer command that specifies columns in a report for which to calculate and print totals. TOTAL returns you to the first SELECT output row.

```
TOTAL column [, column ] ... ;
```

column

identifies a column to total. The column must be a column with a numeric data type from the current detail line and cannot be an IF/THEN/ELSE column. You can specify *column* as a column name, an alias, a detail alias, or COL *number* (which specifies the position of the column in the select list).

Considerations—TOTAL

- Only one TOTAL command is in effect at a time. When you enter a TOTAL command, it replaces the previous TOTAL command.
- A total prints immediately beneath the print item it totals and uses the same display format as the print item.

When you specify a print item in a DETAIL command, make sure the display format in the AS clause allows enough room for the total. If the total is too large for its display format, the field is filled with overflow characters instead of a total.

To include a date-time value in an AS clause, use JULIANTIMESTAMP to convert it to a Julian timestamp.

To define a sufficiently large total field for an INTERVAL total (which cannot use an AS clause), specify a heading wide enough for the largest total value you expect.

Totals require three report lines: two for underline characters (see [UNDERLINE_CHAR Option](#) on page U-1) and one for the totals.

- In calculating totals, the report writer uses the maximum format for the item's data type and the same scale as the item to be totaled. In unusual cases (such as when an expression contains an item multiplied by an extremely small fractional value), this strategy can cause numeric overflow.

Specifying small numeric values in exponential notation (for example, .0000246615 E0 instead of .0000246615) can prevent overflow by causing the report writer to use a floating-point format for such calculations.

Examples—TOTAL

- This example selects data, specifies a detail line, and specifies a column to total:

```
>> SELECT * FROM PERSONL.EMPLOYEE WHERE DEPTNUM = 3000;
S> DETAIL EMPNUM, SALARY;
S> TOTAL SALARY;
```

The output might look like:

EMPNUM	SALARY
-----	-----
1	175500.00
	...

	489075.00

- This example selects data, uses NAME to assign names to the last two columns in the report, and then specifies totals for those columns:

```
>> SELECT PARTNUM, AVG(QTY_ORDERED),
AVG(UNIT_PRICE*QTY_ORDERED)
+> FROM SALES.ODETAIL
+> GROUP BY 1
+> ORDER BY 1;
S> NAME COL 2 AVGORD;
S> NAME COL 3 AVGPR;
S> TOTAL AVGORD, AVGPR;
```

The output might look like:

PARTNUM	AVGORD	AVGPR
-----	-----	-----
	...	
7102	55	687.50
7301	3	4807.99
	-----	-----
	-----	-----
	1480	55683.59

TRANSIDS Table

The TRANSIDS table is a catalog table used to prevent multiple DDL operations from being executed on the same catalog at the same time under the same TMF transaction. [Table T-2](#) describes the contents of the TRANSIDS table.

The TRANSIDS table was created in version 1.

Table T-2. TRANSIDS Table

Column Name	Data Type	Description
1 TRANSID*	LARGEINT SIGNED	TMF transaction ID under which currently executing DDL operation that uses this catalog is being performed
2 SYSNUMBER	SMALLINT SIGNED	Node number of catalog manager process executing the DDL statement
3 PROCESSNAME	CHAR(6)	Process name of catalog manager process executing the DDL statement

* Indicates primary key

TRIM Function

The TRIM function removes leading and trailing characters from a character string.

```

TRIM ( [ [ trim-type ] [ trim-char ]
        FROM ] character-string )

where trim-type is:

    { LEADING      }
    { TRAILING     }
    { BOTH         }

trim-char and character-string are:

    { string-literal      }
    { column-name        }
    { param-name         }
    { host-var-name      }
    { UPSHIFT function   }
    { character-expression }

```

trim-type

specifies whether characters are to be trimmed from the leading end (LEADING), trailing end (TRAILING), or both ends (BOTH) of the character string. If you omit *trim-type*, the default is BOTH.

trim-char

specifies the character to be trimmed from the string. The data type of *trim-char* must be CHARACTER with a maximum length of 1. If you omit *trim-char*, SQL trims blanks (" ") from the string.

character-string

specifies the string from which to trim characters.

Consideration—TRIM Function

The result is always of type VARCHAR, with the same collating attributes as the source *character-string*. If the source character string is an upshifted CHAR or VARCHAR string, the result is an upshifted VARCHAR type. The TRIM character and the character string to be trimmed should have the same or comparable collations and identical character sets. Otherwise, SQL returns an error.

Examples—TRIM Function

- This example returns "Robert":

```
TRIM ("    Robert    ")
```

- This example uses a table created:

```
CREATE TABLE NAMES (FIRST_NAME CHAR(15), LAST_NAME CHAR(15))
INSERT INTO NAMES VALUES ("Robert", "Smith")
```

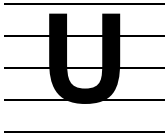
- To retrieve "Robert Smith" without extra blanks, you could use the TRIM operator:

```
TRIM (TRAILING " " FROM FIRST_NAME) || " " || TRIM
(LAST_NAME)
```

For more information about the concatenation operator (||), see [Character Expressions](#) on page C-14.

- You can also use the TRIM function to perform a LIKE comparison with fixed-length host variables:

```
CREATE TABLE T (A CHAR(10))
INSERT INTO T VALUES ("ROBERT")
INSERT INTO T VALUES ("ROMEO")
INSERT INTO T VALUES ("JOHN")
SELECT A FROM T WHERE TRIM(A) LIKE :host_var
```



UNDERLINE_CHAR Option

UNDERLINE_CHAR is an option of the SQLCI report writer SET STYLE command that specifies the character to use for underlining.

Underline characters print below headings and above subtotals and totals.

```
UNDERLINE_CHAR "character"
```

character

is a printable, single-byte character to use for underlining. The default is - (hyphen).

Example—UNDERLINE_CHAR

This example changes the underline character to the equal sign:

```
>> SET STYLE UNDERLINE_CHAR "=" ;
```

A heading appears as:

```
ORDERNUM  
=====
```

UNLOCK TABLE Statement

UNLOCK TABLE is a DCL statement that releases locks owned by SQLCI or by a host program on a nonaudited table or on underlying nonaudited tables of a view.

UNLOCK TABLE does not affect audited tables. (Ending a TMF transaction unlocks an audited table. For more information, see [TMF Transactions](#) on page T-6.)

```
UNLOCK TABLE { name }
```

name

is the name of a table or view to unlock.

Considerations—UNLOCK TABLE

- To unlock a table, you must have authority to read the table. To unlock a view, you must have authority to read the tables underlying the view.
- Always follow an UNLOCK TABLE statement with a CONTROL TABLE statement with the TABLELOCK ENABLE option, as shown in the examples. The CONTROL TABLE statement provides information at compile time, unlike the UNLOCK TABLE statement, which is in effect only at execution time.

When you specify the default value, TABLELOCK ENABLE, the executor can determine at run time whether a table lock is necessary. This strategy increases concurrency for statements that do not require a table lock, such as those statements that need only a few row locks.

- A host program cannot execute an UNLOCK TABLE statement if the program has a cursor open on the table or view with STABLE or REPEATABLE access. Close the cursor with a CLOSE or FREE RESOURCES statement before you execute UNLOCK TABLE statement.

Examples—UNLOCK TABLE

- This example locks and unlocks a nonaudited table from an SQLCI session in which the AUTOWORK AUDITONLY is ON:

```
>> VOLUME $VOL1.PERSNL;
>> LOCK TABLE JOB IN EXCLUSIVE MODE;
--- SQL operation complete
>> DELETE FROM JOB WHERE JOBCODE
+> NOT IN (SELECT DISTINCT JOBCODE FROM EMPLOYEE);
--- 8 row(s) deleted.
...
>> UNLOCK TABLE JOB;
--- SQL operation complete
```

- This example locks and unlocks a nonaudited table from a program:

```
EXEC SQL
    LOCK TABLE SALES.PARTS IN EXCLUSIVE MODE;
EXEC SQL
    CONTROL TABLE SALES.PARTS TABLELOCK ON;
...
EXEC SQL
    UNLOCK TABLE SALES.PARTS;
EXEC SQL
    CONTROL TABLE SALES.PARTS TABLELOCK ENABLE;
```

UPDATE Statement

UPDATE is a DML statement that updates rows of a table or protection view.

```
UPDATE { name } SET col = exp [, col = exp ] ...
```

```

[ [ | WHERE search-cond | ] ]
[ [ | [ FOR ] { STABLE | REPEATABLE } ACCESS | ] ]
[ [ | WHERE CURRENT OF cursor | ] ]

```

name

is the name (or an equivalent DEFINE) of the table or protection view to update.
name cannot be the name of a catalog table.

col

is the name of the column to update. You cannot qualify the column name, repeat a column name, or specify the column of a primary key or clustering key.

exp

is an SQL expression that specifies a value for the column. *exp* can be a literal, the keyword NULL (which indicates a null value), a host variable (possibly with an indicator variable to specify a null value), or a parameter, but its data type (including character set, if any) must be the same as that associated with the column.

exp cannot include a subquery or an aggregate function, but it can refer to any column in the row, including the SYSKEY column. (If *exp* refers to the column being updated, SQL uses the prior value to evaluate the expression and determine the new value.)

WHERE *search-cond*

specifies a search condition for selecting rows to update. Subqueries within *search-cond* cannot refer to the table or view being updated. For more information, see [Search Conditions](#) on page S-5.

[FOR] { STABLE | REPEATABLE } ACCESS

specifies the degree of consistency required while rows are compared to the search condition; affects the type of locks SQL uses in the execution plan for the UPDATE and the degree of concurrency for other applications:

STABLE	specifies stable consistency; SQL uses locks of short duration.
REPEATABLE	specifies repeatable consistency; SQL uses locks of long duration.

The default is REPEATABLE.

For a detailed discussion of both STABLE and REPEATABLE, see [Access Options](#) on page A-1.

WHERE CURRENT OF *cursor*

(for use in programs only) specifies a host-program cursor to identify the row to be updated. The cursor must be positioned at a single row to be updated and not between rows.

For static SQL programs, each column to be updated must appear in the FOR UPDATE clause of the cursor declaration. For dynamic SQL programs, each column to be updated must appear in the FOR UPDATE clause of the SELECT that defines the cursor.

You cannot specify WHERE CURRENT OF *cursor* when using parallel execution.

Considerations—UPDATE

- UPDATE requires authority to read and write to the table or view being updated and authority to read any table or view specified in subqueries of the search condition.

You cannot use UPDATE to update a catalog table.

- Rows must be locked to be updated. The condition that identifies the row to be updated determines the locking protocol for the update. If a cursor identifies the row, the ACCESS option of DECLARE CURSOR determines the locking protocol; if a WHERE clause identifies the row, the access option in the UPDATE statement determines the locking protocol.
- Each updated row must satisfy the assertions of the table or underlying table of the view. An updated row from a protection view created with the WITH CHECK OPTION must satisfy the view selection criteria. (The selection criteria are specified in the WHERE clause of the AS *select-statement* clause in the CREATE VIEW statement.) No column updates occur unless all these conditions are satisfied.
- Rows are updated in sequence. If an error occurs, SQL returns an error message and stops updating the table or view.

When the UPDATE finishes successfully, SQL reports the number of times rows were updated during the operation.

Under certain conditions, updating a table with indexes causes SQL to update the same row more than once, causing the number of updates that SQL reports to be higher than the actual number of rows changed. The data in the table is correct and the message correctly reports the number of times rows were updated. This behavior (a variation of the “Halloween problem” described in computer science literature) occurs when all these are true:

- The optimizer chooses the alternate index as the access path.
- The index columns in the equal-predicate are not changed by the update.
- Another index column of the same index is updated to a higher value (if the column is stored in ASCENDING order, or a lower value if the column is stored in DESCENDING order).
- For a fixed-length character column, an update value shorter than the column length is padded with single-byte ASCII blanks (HEX20) to fill the column.

For a varying-length character column, an update value is not padded; its length is the length of the value specified. In an entry-sequenced table, a value that updates a varying-length character column must be the same length as the value it replaces.

- For information on buffering UPDATE operations, see [CONTROL TABLE Directive](#) on page C-77.
- These guidelines apply specifically to using the UPDATE statement in host programs:
 - A TMF transaction must be in progress if you use the WHERE CURRENT OF clause to update an audited table or view. The same TMF transaction must include the OPEN cursor, FETCH, and UPDATE operations.
 - When using an SQL cursor in a host language program, an UPDATE WHERE CURRENT operation provides a performance benefit over a stand-alone UPDATE operation. Updating through a cursor uses virtual sequential block buffering (VSBB) unless another cursor or a stand-alone UPDATE or DELETE operation for the same table is used within the same process.

Use an UPDATE WHERE CURRENT operation instead of a stand-alone operation to access a table (directly or through a view) within the same process whenever possible. Using a stand-alone operation or another cursor to access a table (directly or through a view) within the same process invalidates VSBB. Invalidating VSBB can degrade performance substantially.

- SQL returns these values to the SQLCODE variable after a DELETE:

0	The UPDATE succeeded
100	No rows satisfied the search condition
> 0	A warning was issued
< 0	An error occurred; the UPDATE did not complete

The SQLCA records the number of rows updated.

Examples—UPDATE

- This example updates a single row of the ORDERS table that contains information about order number 200038 and changes the delivery date:

```
>> UPDATE SALES.ORDERS SET DELIV_DATE = 880522
+> WHERE ORDERNUM = 200038;
--- 1 row(s) updated.
```

- This example updates several rows of the CUSTOMER table:

```
>> UPDATE SALES.CUSTOMER SET CREDIT = "A1"
+> WHERE CUSTNUM IN (21, 3333, 324);
--- 3 row(s) updated.
```

- This example increases the salary of each employee working for a department located in San Francisco. The subquery is evaluated for each row of the DEPT table and returns department numbers for departments located in San Francisco.

```
>> VOLUME $VOL1.PERSNL;
>> UPDATE EMPLOYEE SET SALARY = SALARY * 1.1
+> WHERE DEPTNUM IN (SELECT DEPTNUM FROM DEPT
+> WHERE LOCATION = "SAN FRANCISCO" );
```

- Suppose that you want to change an employee's number. The employee is the manager of a department. Because EMPNUM is a primary key of the EMPLOYEE table, you must delete the employee's record and insert a record with the new number.

You must also update the DEPT table to change the MANAGER column to the employee's new number. To ensure that all your changes take place (or that none of them do), you should perform the operation as a TMF transaction, as shown:

```
>> VOLUME $VOL.PERSNL;
>> BEGIN WORK;
>> DELETE FROM EMPLOYEE WHERE EMPNUM = 23;
--- 1 row(s) deleted.
>> INSERT INTO EMPLOYEE
+> VALUES (50, "JERRY", "HOWARD", 1000, 100, 137000.10);
--- 1 row(s) inserted.
>> UPDATE DEPT SET MANAGER = 50 WHERE DEPTNUM = 1000;
--- 1 row(s) updated.
>> COMMIT WORK;
```


UPDATE STATISTICS Statement

UPDATE STATISTICS is a DDL statement that updates the statistics stored in the catalog for the specified table. SQL does not automatically update statistics; you must execute this statement to have current statistics in your catalog.

```
UPDATE [ ALL ] STATISTICS FOR TABLE table

      [ RECOMPILE      ] [ SIMPLE          ] [ EXACT              ]
      [ NO RECOMPILE   ] [ PROBABILISTIC ] [ SAMPLE n BLOCKS ]
```

ALL

requests updated statistics for all columns in the table.

If you do not specify ALL, statistics are updated only for columns that make up the primary or clustering key of the table and columns specified in any index on the table.

table

is the name of the table for which to update statistics.

RECOMPILE | NO RECOMPILE

specifies whether to invalidate program files that use the table affected by the UPDATE STATISTICS operation:

RECOMPILE invalidates program files, but if *table* has SIMILARITY CHECK ENABLED, does not invalidate files compiled with CHECK INOPERABLE PLANS.

NO RECOMPILE does not invalidate the program files.

RECOMPILE is the default behavior. SQL uses the RECOMPILE option if you do not specify either option.

Only recompiled programs can take advantage of the new statistics. Invalidating programs forces recompilation unless the programs were compiled with the CHECK INOPERABLE PLANS option. SQL automatically recompiles an invalid program at execution time (unless you specified options to restrict recompilation when you originally compiled it), but you need to explicitly recompile each program to revalidate it.

If you specify RECOMPILE, you might want to use VERIFY to display the names of invalid programs. See [VERIFY Command](#) on page V-2.

SIMPLE | PROBABILISTIC

specifies whether to compute statistics using a new internal algorithm, called PROBABILISTIC, or the algorithm used in previous RVUs of NonStop SQL/MP, called SIMPLE.

The PROBABILISTIC algorithm is designed to give more accurate results than the previous (SIMPLE) algorithm. Moreover, when you specify the PROBABILISTIC option, SQL computes statistics in parallel on partitioned tables; SQL operates in parallel on each partition in a table.

If you specify the PROBABILISTIC option, SQL ignores the EXACT and SAMPLE n BLOCKS options. With the PROBABILISTIC algorithm, SQL always reads every row in the table. If you specify the SIMPLE option, SQL uses either the EXACT or the SAMPLE n BLOCKS option, depending on which you specify.

The default option is the SIMPLE algorithm.

In future product version updates (PVUs) of NonStop SQL/MP, SQL might switch to the PROBABILISTIC option as the default option. When this change is incorporated, an UPDATE STATISTICS statement that has no options specified will no longer use the SIMPLE algorithm, but instead use the PROBABILISTIC algorithm. At present, if you want to use the PROBABILISTIC algorithm, you must specify the PROBABILISTIC option in your UPDATE STATISTICS statements in embedded SQL or OBEY command files.

EXACT | SAMPLE n BLOCKS

specifies whether to compute statistics by reading each row in each partition of the table (EXACT) or to compute statistics by sampling n blocks of each partition in the table (SAMPLE n BLOCKS) and extrapolating from that sample. The value n must be greater than zero.

If you do not specify either option, SQL computes statistics based on reading all rows in partitions smaller than 1000 blocks and reading approximately 500 blocks from partitions of 1000 blocks or more. (In the latter case, SQL reads a larger sample if less than 3 percent of the total values have been sampled and 97 percent or more of the sampled values are distinct.)

Considerations—UPDATE STATISTICS

- To update statistics for a table, you must be the generalized owner of the table. You must also have authority to read the table and to write to the catalogs that describe the table.

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 execute an UPDATE STATISTICS statement while another process is executing a DDL operation on the same object. The specific error depends on the DDL operation involved and the phase of the operation at which the conflict occurs. For more information, see [Concurrency](#) on page C-65.

- UPDATE STATISTICS collects and saves these statistics:
 - Date and time UPDATE STATISTICS was last run on the table
 - Number of rows in the table

- Byte address of the end-of-file of the table
(The number of bytes indicates the space used by the table.)
- Percent of blocks that contain rows (nonempty blocks)
- Number of index levels for the indexes on the table
(Each level represents a disk access operation required to retrieve data.)
- Number of unique entries in each column
(All null entries count as one unique entry, just as with other values.)
- Second highest value in each column
(Ignores null values; uses highest nonnull value if number of unique entries is three or fewer.)
- Second-lowest value in each column
(Ignores null values; uses lowest nonnull value if number of unique entries is three or fewer.)

Except for row count, statistics are for the entire table (not for individual partitions as in earlier PVUs of NonStop SQL/MP). Note that the second-highest and second-lowest values reflect the column values of the entire table, rather than of a single partition. The unique entry count is the unique entry count for the entire table divided by the number of partitions in the table.

UPDATE STATISTICS does not erase statistics already in the catalog that are unaffected by the current update. For example, if an UPDATE STATISTICS statement follows an UPDATE ALL STATISTICS statement for the same table, the statistics remain unchanged for the columns that are not part of an index or the primary key or clustering key.

When a table or a table partition is empty, UPDATE STATISTICS uses default values to update items for which it does not have actual values and returns warning 1404. UPDATE STATISTICS always enters the current timestamp in the STATISTICSTIME column of the BASETABS catalog table, whether or not other statistics are available.

These statistics are used by the SQL compiler in selecting a strategy for an execution plan. If the structure, contents, or indexes of the table have changed, or if you experience a degradation in performance, use UPDATE STATISTICS to update the information about the table in the catalog. You should also recompile your programs. With up-to-date information, the SQL compiler can choose the best path for queries on the table.

- Before executing UPDATE STATISTICS, determine whether the table is fragmented (that is, whether the pages have a large amount of unused space). At the command interpreter prompt, use the FILEINFO command with the STATISTICS option to check the table. If the average percent of slack is large, use

SQLCI LOAD or FUP RELOAD to reorganize the table before updating the statistics.

- UPDATE STATISTICS momentarily locks the definition of the table in the catalog during the operation, but not the table itself; the statement uses BROWSE ACCESS.
- All partitions of a table must be available for SQL to generate accurate statistics, not just the partition specified in the UPDATE STATISTICS statement.

If any partitions are unavailable when you run UPDATE STATISTICS, SQL issues a warning, skips the unavailable partitions, and updates statistics for the remaining partitions using default values for the unavailable partitions. If this occurs, you should rerun UPDATE STATISTICS with all partitions as soon as possible.

If a partitioned table does not have statistics (that is, if you have never executed UPDATE STATISTICS for that table) and one of the partitions is not available when you execute SELECT or another DML operation on the table, SQL returns a warning and a file-system error even if the query does not retrieve any rows from the unavailable partition. The warning does not occur for very small tables, but you can prevent it from occurring at all by executing UPDATE STATISTICS at least once for any partitioned table.

- When your table has many partitions—for example, 100—you might want to avoid putting the UPDATE STATISTICS statement in a user-defined TMF transaction. With many partitions, the UPDATE STATISTICS operation might take so long that TMF might have too little log file space to perform all the logging required by other TMF transactions at the time.

If no user-defined TMF transaction is in progress when UPDATE STATISTICS executes, SQL starts several for the operation but does not include scanning the table for information within a transaction. Because UPDATE STATISTICS uses BROWSE ACCESS to scan the table, the results are approximate.

- To see current statistics, use the SELECT statement to retrieve the information from the catalog tables. For example, the second-highest value of a column is stored in the COLUMNS table. The STATISTICSTIME column of the BASETABS table contains the time that statistics for the table were last updated, stored as a timestamp in Greenwich mean time.

For more information about the information stored in the catalog, see [CATALOGS Table](#) on page C-11 or the entry for a specific catalog table.

Examples—UPDATE STATISTICS

- This example updates all statistics for a table named PARTLOC on the current default volume and subvolume without invalidating the associated programs:

```
UPDATE ALL STATISTICS FOR TABLE PARTLOC NO RECOMPILE;
```

- This example updates statistics for columns in the primary key or in indexes for a table named STUDENTS, directing SQL to calculate the statistics based on the contents of the first 300 blocks in each partition:

```
UPDATE STATISTICS FOR TABLE STUDENTS SAMPLE 300 BLOCKS;
```

UPGRADE CATALOG Command

UPGRADE CATALOG is an SQLCI utility command that converts catalogs to a newer version so the catalogs can register objects associated with a newer version of the SQL/MP software.

```
UPGRADE CATALOG[S] [ catalogs ] [ TO version ] ;
```

catalogs

specifies the catalogs to upgrade. It can be a single catalog name, a type CATALOG DEFINE name, or a name that specifies multiple catalogs by including these wild-card characters:

?	matches any single character
*	matches 0 to 8 characters

For example, these names specify multiple catalogs:

MYCAT?	matches MYCAT1, MYCAT2, and MYCATX (and possibly others), but not MYCAT48
\$DATA.*	matches all catalogs on volume \$DATA
=APPCAT	matches the catalog specified by the =APPCAT DEFINE
(MYCAT?, =APPCAT)	matches MYCAT1, MYCAT2, MYCATX, (and possibly others) and the catalog specified by =APPCAT
\$*.*	matches all catalogs on the current default node except the system catalog

Catalogs specified by *catalogs* can be either local or remote, but cannot be system catalogs. (Use UPGRADE SYSTEM CATALOG to convert a system catalog.)

The default is the current default catalog.

If SMF is installed on your node, *catalogs* cannot specify any catalog on a \$*.ZYS*. subvolume.

TO *version*

specifies the catalog format version for the upgraded catalog.

You can express *version* as an integer (2, 300, 310, 315, 320, 325, or 330) or as a string (A011, A300, A310, A315, A320, A325, or A330), but the version you

specify must be newer than the current version of the catalogs you specify with *catalogs*. In addition, *version* must not specify a version newer than the version of the SQL/MP software executing UPGRADE CATALOG or newer than the version of the SQL/MP software running on the node of the catalog being upgraded.

The default is the version of the SQL/MP software installed on the node on which the catalogs reside, or the version of the SQL/MP software executing UPGRADE CATALOG, whichever is older.

For more information about NonStop SQL/MP product versions, see [Versions](#) on page V-6 or the *SQL/MP Version Management Guide*.

Considerations—UPGRADE CATALOG

- To upgrade a catalog, you must be a generalized owner of the catalog. You must also have authority to write to the CATALOGS table in the system catalog.

UPGRADE CATALOG requires exclusive access to the catalogs being upgraded. Other processes cannot access the catalogs during the upgrade. The upgrade fails if another process has one of the tables in the catalogs open when you execute UPGRADE CATALOG. In addition, all indexes, views, and programs registered in the catalogs must be available for read access.

For performance reasons, SQLCI sometimes keeps catalog files open for five minutes after the SQLCI command or statement that uses them finishes. This strategy can interfere with a subsequent UPGRADE CATALOG operation in the SQLCI session. If such interference occurs, exit SQLCI and start a new SQLCI session.

- Unless UPGRADE CATALOG executes within a user-defined transaction, an error that causes the upgrade of one catalog specified in *catalogs* to fail does not necessarily cause the upgrades of other catalogs specified in *catalogs* to fail. (Use GET VERSION if you want to check the version of a specific catalog.)
- UPGRADE CATALOG invalidates any program that refers to a catalog table in the upgraded catalogs, but does not invalidate a program merely because it is registered in an upgraded catalog or because it accesses objects registered in an upgraded catalog.

Examples—UPGRADE CATALOG

- This example converts the current default catalog to the same version as the current SQL/MP software:

```
>> UPGRADE CATALOG;
```

- Either of these commands converts the catalog on subvolume \$VOL.SVOL to version 320:

```
>> UPGRADE CATALOG $VOL.SVOL TO 320;
>> UPGRADE CATALOG $VOL.SVOL TO A320;
```

- This example converts all catalogs (except the system catalog) on the current default node to the same version as the current SQL/MP software:

```
>> UPGRADE CATALOGS $*.*;
```

UPGRADE SYSTEM CATALOG Command

UPGRADE SYSTEM CATALOG is an SQLCI utility command that allows a user with super ID authority to convert the system catalog on the local node to support a newer version of NonStop SQL/MP.

```
UPGRADE SYSTEM CATALOG [ TO version ] ;
```

TO version

specifies the catalog format version for the upgraded system catalog.

You can express *version* as an integer (2, 300, 310, 315, 320, 325, or 330) or as a string (A011, A300, A310, A315, A320, A325, or A330), but the version you specify must be greater than the current version of the catalogs you specify with *catalogs*. In addition, *version* must not be the product version of the SQL/MP software installed on the node.

The default is the version of the SQL/MP software installed on the node.

For more information about NonStop SQL/MP product versions, see [Versions](#) on page V-6 or the *SQL/MP Version Management Guide*.

Considerations—UPGRADE SYSTEM CATALOG

- Only the local super ID can execute UPGRADE SYSTEM CATALOG.
UPGRADE SYSTEM CATALOG requires exclusive access to the system catalog being upgraded. Other processes cannot access the system catalog during the upgrade. The upgrade fails if other processes have the system catalog open when you issue the UPGRADE SYSTEM CATALOG command. In addition, all indexes, views, and programs registered in the system catalog must be available for read access.
- UPGRADE CATALOG invalidates any program that refers to a table in the system catalog, but does not invalidate a program merely because it is registered in the system catalog or because it accesses objects registered in the system catalog.

Example—UPGRADE SYSTEM CATALOG

This example upgrades the system catalog on the local system to version 315:

```
>> UPGRADE SYSTEM CATALOG TO 315;
```

UPSHIFT Function

UPSHIFT is a function that upshifts single-byte characters. UPSHIFT can appear in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or qualifying a new value in an UPDATE or INSERT statement.

UPSHIFT returns a string of either character or varying-length character data, depending on the data type of the input string.

For more information, see [Character Expressions](#) on page C-14.

UPSHIFT (<i>character-exp</i>)

character-exp

is a character expression that specifies a string of single-byte characters to upshift.

Considerations—UPSHIFT

- If the character expression you specify as an argument to UPSHIFT is associated with a collation, SQL upshifts the string based on the rules specified in that collation.

If the argument is not associated with a collation, SQL converts lowercase characters to uppercase characters according to the usual rules of English.

- In version 2, the argument to UPSHIFT could be a column name, a string literal (without an associated character set), a parameter name, or a host variable name, but could not be any other form of character expression.

If you use the UPSHIFT function according to version 2 rules, the program will be at least version 2. If you use the UPSHIFT function according to version 3 rules, the program will be at least version 3.

Examples—UPSHIFT

- This example selects all values from the column CUSTNAME and returns them in uppercase:

```
>> SELECT UPSHIFT(CUSTNAME) FROM =CUSTOMER;
```

- This example performs a case-insensitive search for the customer name HOTSYS. (In the table, the name can be in lowercase, uppercase, or mixed case.)

```
>> SELECT * FROM =CUSTOMER
+> WHERE UPSHIFT(CUSTNAME) = "HOTSYS";
```

- This example returns all rows from two tables in which department names have the same value, regardless of case:

```
>> SELECT * FROM =DEPT1, =DEPT2
+> WHERE UPSHIFT(DEPT1.DEPTNAME) = UPSHIFT(DEPT2.DEPTNAME);
```


USAGES Table

The USAGES table is a catalog table that keeps records of dependencies between objects and between programs and objects. [Table U-1](#) describes the contents of the USAGES table. In the column descriptions, the terms *initial object* and *used object* refer to an object on which another object depends. The dependent object is called a *using object*.

Table U-1. The USAGES Table

Column Name	Data Type	Description
1 USEDOBJNAME *	CHAR(34)	Name of initial object
2 USEDOBJTYPE *	CHAR(2)	CP if collation IN if index TA if table VI if view
3 RELATIONSHIPTYPE *	CHAR(2)	DP if using object depends on used object
4 USINGOBJNAME *	CHAR(34)	Name of dependent object
5 USINGOBJTYPE *	CHAR(2)	IN if index VI if view PG if program TA if table
6 USEDATALOGNAME	CHAR(25)	Subvolume of catalog that describes initial object
7 USINGCATALOGNAME	CHAR(25)	Subvolume of catalog that describes dependent object

* Indicates primary key

The USAGES table was created in version 1. The CP option was added to USEDOBJTYPE and USINGOBJTYPE in version 300.

The relations recorded in the USAGES table are:

- Program uses view, table, or index
- View uses view or table
- Index uses table
- Table, view, index, or program uses collation

If a table or index is partitioned or a view has an underlying partitioned table, the relations are established with the primary partition and recorded in the primary partition's catalog.

Relations between objects that reside in separate catalogs appear in both catalogs.

Relations between partitions and relations to constraints, comments, or files are not recorded. This information is already present in other catalogs.

Only direct relations are stored. Indirect relations can be determined by following a path from one relation to another.

The lock length for the USAGES table is set to zero to indicate that the entire primary key length should be used for locking.

Guardian names in USAGES are fully qualified and use uppercase characters. Names of SQL programs in OSS files are stored as the corresponding ZYQ Guardian names, not OSS pathnames.

User-Defined Keys

A user-defined primary key is made up of the columns specified in the PRIMARY KEY clause of the CREATE TABLE statement when the table is created. Values for a user-defined primary key must be unique within the table. Only a table stored in a key-sequenced file can have a user-defined primary key.

Each column in the primary key has an ordering characteristic (either ascending or descending) that determines the order for storing and retrieving the rows. If multiple rows share the same value for the first column of the primary key, the value and the ordering characteristic of the second column determines the order for storing or retrieving the rows, and so forth.

The columns in a primary key cannot contain null values and cannot be updated.

The length of a primary key cannot exceed 255 bytes. To calculate the length of the key, add the number of bytes in the columns that make up the key. For each varying-length character column, add 2 to the byte count determined by the number and size of the characters allowed in the column. (SQL uses the two additional bytes to store the character count.)

The system does not generate a SYSKEY for a table that has a user-defined primary key.

Utilities

SQLCI includes a variety of utilities that perform database maintenance tasks and that access editors and other commonly used Guardian utilities from within an SQLCI session.

These SQLCI commands access SQLCI utilities:

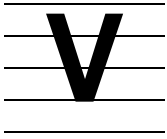
CLEANUP	INVOKE
CONVERT	LOAD
COPY	MODIFY CATALOG
DISPLAY USE OF	MODIFY LABEL
DOWNGRADE CATALOG	MODIFY REGISTER
DOWNGRADE SYSTEM CATALOG	PERUSE

DUP	PURGE
EDIT	PURGEDATA
EXPLAIN	SECURE
FILEINFO	TEDIT
FILENAMES	UPGRADE CATALOG
FILES	UPGRADE SYSTEM CATALOG
FUP	VERIFY

The SQLCI commands EDIT, FUP, PERUSE, and TEDIT access Guardian utilities from SQLCI that you can also access from TACL; the other utilities are part of NonStop SQL/MP. For more information about these utilities, see entries for specific commands.

You can also use the Guardian utilities, BACKUP, DCOM, DSAP, PUP, and RESTORE on SQL/MP objects, but you must run these utilities from TACL, not from SQLCI. Guardian utilities are described in the *Guardian Disk and Tape Utilities Reference Manual* and the *Peripheral Utility Program (PUP) Reference Manual*.

One additional SQL utility, GOAWAY, can be run only from TACL. GOAWAY enables a user with super ID authority to delete SQL files or shadow labels that cannot be removed with other commands or utilities. For more information, see [GOAWAY Command](#) on page G-6.



VARCHAR_WIDTH Option

VARCHAR_WIDTH is an option of the SQLCI report writer SET STYLE command that specifies the maximum number of single-byte characters the report writer can display in a print item of a varying-length character data type.

<code>VARCHAR_WIDTH <i>number</i></code>
--

number

is an integer in the range 1 through 255 that specifies the maximum number of single-byte characters that can appear in the print item. The default is 80.

Consideration—VARCHAR_WIDTH

You can override the VARCHAR_WIDTH setting by using the Cn display descriptor in an AS clause for a print item. For information, see [AS Clause](#) on page A-60.

You must override VARCHAR_WIDTH to print varying-length character items with more than 255 bytes. For example, you can use AS C0.40 to print a VARCHAR item that contains 1000 single-byte characters (or 500 multibyte characters). The value prints on multiple lines in 40-byte fields.

Example—VARCHAR_WIDTH

This example sets a VARCHAR_WIDTH of 60 (60 single-byte characters or 30 multibyte characters):

```
>> SET STYLE VARCHAR_WIDTH 60;
```

VERIFIEDWRITES File Attribute

VERIFIEDWRITES is a file attribute that specifies or inhibits a verification disk read after each disk write. VERIFIEDWRITES applies to key-sequenced, relative, and entry-sequenced files, in addition to indexes.

<code>{ VERIFIEDWRITES NO VERIFIEDWRITES }</code>

The table default is NO VERIFIEDWRITES. The index default is its table's value at index creation.

Consideration—VERIFIEDWRITES

On a verified write, the disk controller hardware makes a byte-by-byte comparison between the newly written data and the corresponding data in the controller's memory. Verified writes help ensure the accuracy of write operations, but they increase response time and disk utilization.

VERIFY Command

VERIFY is an SQLCI utility command that reports whether SQL objects and programs are consistently described in the file labels and the catalog.

VERIFY also lists invalid SQL programs described in specified catalogs and optionally generates a command file to recompile the invalid programs stored on Guardian files. (The command file does not include commands to recompile invalid programs stored on OSS files.)

```
VERIFY [ DEF[inition] [OF] ] qualified-fileset-list
      [ [,] SOURCE edit-file [ CLEAR ] ][REPAIR];
```

[DEF[inition] [OF]]

is an optional phrase that does not change the meaning of the command. The words DEFINITION and OF are reserved words.

qualified-fileset-list

is a qualified fileset list that specifies SQL objects and programs to verify, either directly or indirectly (by specifying catalogs or subvolumes that include the objects and programs). For information, see [Qualified Fileset List](#) on page Q-1.

If *qualified-fileset-list* includes a FROM CATALOGS clause, VERIFY expands a list containing wild-card characters using the specified catalogs; if you omit FROM CATALOGS, VERIFY expands the list using the Guardian disk directory.

To list invalid programs only, use *qualified-fileset-list* with the WHERE SQLPROGRAM qualifier.

If SMF is installed on your node, *qualified-fileset-list* cannot specify any file, program, or object on a \$*.ZYS*. subvolume.

SOURCE *edit-file* [CLEAR]

directs SQL to generate SQLCOMP commands to recompile any invalid SQL programs on Guardian files included in *qualified-fileset-list* and to write the SQLCOMP commands in an EDIT file named *edit-file*. If *edit-file* does not exist, SQL creates it.

You can use *edit-file* with the TACL OBEY command to recompile the invalid SQL programs in the Guardian files. (SQL does not generate commands to recompile SQL programs in OSS files.)

CLEAR clears *edit-file* before writing SQLCOMP commands. If you omit CLEAR, SQL appends commands to the existing text.

If SMF is installed on your node, *edit-file* must be either a logical or direct file.

[REPAIR]

directs VERIFY to re-create any missing SQL object using information available in the catalog. The newly created object does not contain any data. The VERIFY command with this option creates the physical object and checks the object for definitional integrity. This option provides support for tables, partitions, views, and indexes.

Considerations—VERIFY

- VERIFY requires read authority for the objects being verified and for related objects and catalogs.

VERIFY requests REPEATABLE access to object definitions in catalogs to prevent DDL operations on the object during the VERIFY operation. If VERIFY cannot attain REPEATABLE access, it issues a warning message and continues using BROWSE access.

VERIFY also requests a SHARED lock on the file label of the object (including all partitions of partition objects). If VERIFY cannot obtain the lock, it issues a warning message and continues without the lock.

- VERIFY checks for definitional integrity, but not for data integrity. An object or program has definitional integrity if its description in the file label is consistent with its definition in the catalog, and the descriptions of related objects in related disk file labels are valid. Specific checks depend on the type of item being checked.

For all objects and SQL programs in Guardian files, VERIFY checks that the version is the same in the file label and in the catalog. VERIFY also checks related catalogs for relationships with other objects as described in the local USAGES table.

For a table, VERIFY checks descriptions of the table and its columns, primary key columns, indexes and index key columns, partitions, and protection views. VERIFY also checks the consistency of index and partition entries in related file labels. However, VERIFY does not check whether the text of a constraint associated with a table is the same in the catalog and in the file label.

For a collation, VERIFY checks descriptions of the collation.

For a protection view, VERIFY checks descriptions of the protection view and its base table and columns. VERIFY also checks the consistency of base table entries in related file labels. However, VERIFY does not check whether the data types of

view columns and the SELECT clause of a protection view are the same in the catalog and in the file label.

For a shorthand view, VERIFY checks descriptions of the view. However, VERIFY does not check whether data types of view columns are the same in the catalog and in the file label.

For an index, VERIFY checks descriptions of the index and its base table, views of the base table, and partitions. VERIFY also checks the consistency of base table and partition entries in related file labels.

For an SQL program, VERIFY checks whether the program is valid or invalid and also checks the consistency of the PCV and PFV values in related file labels. For an SQL program in a Guardian file, VERIFY also checks descriptions of the program.

For a catalog, VERIFY checks descriptions of each catalog table and each SQL-supplied index or protection view on a catalog table.

For partitioned objects, VERIFY makes the checks previously described for each partition of the object. However, VERIFY does not check whether first key values for partitions are the same in the catalog and in the file label.

VERIFY does not check whether a file label is internally consistent with itself.

- If VERIFY locates an invalid object, you might want to change the VALIDDEF flag in the appropriate catalog table to prevent further use of the object until the problem is corrected. (For example, if a view is invalid, change the value of the VALIDDEF column of the VIEWS table to N.) A user with the super ID can change values in a catalog table by using the UPDATE command with a licensed version of SQLCI2. For more information about using a licensed SQLCI2 process, see the *SQL/MP Installation and Management Guide*.
- A program can be invalid for a variety of reasons. In some cases, the appropriate action is to recompile the program. In other cases (especially if the program uses execution-time name resolution), you might want to continue using the invalid program. For more information about changes that can cause program invalidation, see [Program Invalidation](#) on page P-28.

If VERIFY returns SQL error -9853 (a column in the catalog table is corrupted) for a program file, SQL-compiling the program again might correct the problem.

- Avoid using VERIFY within user-defined transactions, because it reduces concurrency for other operations. If a user-defined transaction is not in progress when you execute VERIFY, SQL automatically starts a transaction before verifying each object or program and ends the transaction when the verification of that object or program is complete.

You can use the Break key to interrupt a VERIFY operation. If you roll back the transaction and want to restart the operation, you must issue the VERIFY command again.

Repairing Views

Both the protection view and short hand view, after re-creation, contain a new creation timestamp in the physical objects (file label) and catalog. This invalidates the programs using these views, which might call for recompilation of the programs.

Usage of File Sets With Wild-Card Characters in Repair

File sets used in the VERIFY command, such as \$VOL.SUBVOL.T* cannot be repaired because the filenames are extracted from a disk. The file to be repaired must be fully qualified. File sets with wild-card characters work when the FROM CATALOG option is chosen.

Examples-VERIFY

- This example verifies objects and locates invalid programs in the SALES catalog:

```
>> VERIFY *.*.* FROM CATALOG SALES;
--- Verifying $VOL1.SALES.ASSERTS
--- $VOL1.SALES.ASSERTS Verified.
--- Verifying $VOL1.SALES.BASETABS
--- $VOL1.SALES.BASETABS Verified.
...
(any invalid programs are listed)
...
--- SQL operation complete.
```

- This example verifies the EMPLOYEE table:

```
>> VERIFY $VOL1.PERSNL.EMPLOYEE;
--- Verifying $VOL1.PERSNL.EMPLOYEE
--- $VOL1.PERSNL.EMPLOYEE Verified.
--- SQL operation complete.
```

- This example creates a command file named RECOMPF for recompiling invalid SQL programs in Guardian files in the SALES catalog:

```
>> VERIFY $VOL1.SALES.* FROM CATALOG SALES WHERE SQLPROGRAM
+> SOURCE RECOMPF CLEAR;
...
PROGRAM \SYS.$VOL1.SALES.PROGA
*** WARNING $VOL1.SALES.PROGA is an invalid program.
```

The RECOMPF file contains this command:

```
SQLCOMP /IN \SYS1.$VOL1.SALES.PROGA, OUT file/STOREDEFINES
```

The current SQLCI OUT file is used in the SQLCOMP command.

- This example shows how to use the REPAIR option:

```
>> VERIFY $VOL1.PERSNL.EMPLOYEE REPAIR;
--- Verifying $VOL1.PERSNL.EMPLOYEE
--- $VOL1.PERSNL.EMPLOYEE Verified.
--- SQL operation complete.
```

Versions

A version is associated with each component of SQL/MP software; with each SQL/MP catalog, object, or program; and with each host language compiler that supports NonStop SQL/MP. This table lists NonStop SQL/MP product versions and the RVUs with which they correspond:

SQL/MP Version	RVU
1	C10
2	C30 through D20
300	(Controlled availability release)
310	(Controlled availability release)
315	D30
320	D30
325	N. A.
330	N. A.
335	D42
340	D43
345	D44
350	G06.13

Versions 1, 2, and 315 are sometimes called SQL release 1, SQL release 2, and SQL release 1.0, respectively.

Each new version of NonStop SQL/MP can run all programs that ran on previous versions of NonStop SQL/MP, without recompilation. However, to maintain your programs and take advantage of new features, you should understand the relationships between different versions. Understanding versioning is important if you work on a network that runs different versions of NonStop SQL/MP on different nodes, or if you work with applications and databases that will eventually migrate to newer versions of the SQL/MP software.

This discussion summarizes major rules that govern relationships between SQL/MP items of different product versions. For more information, see the *SQL/MP Version Management Guide*.

SQL/MP Component Versions

Each component of the SQL/MP software has a version. Except for the message file, all SQL/MP software components on any one node in a network must have the same product version. SQL/MP software on different nodes in the same network can have different product versions.

The message file used by SQLCI also has a version. SQLCI can execute with a version of the message file that is older than the version of the other SQL/MP software on the node, but some messages might be missing or inaccurate as a result. (SQLCI issues a warning if you start an SQLCI session that uses an older version of the message file.)

A given version of SQL/MP software can operate on SQL objects registered in catalogs with the same version as the software or in catalogs with an older version than the software. A version of SQL/MP software cannot operate on objects registered in catalogs with a newer version than the software.

The command `GET VERSION OF SYSTEM` returns the version of SQL/MP software installed on a node.

Catalog Versions

Each SQL/MP catalog has a version that indicates the newest-version object you can register in the catalog.

When you create a catalog using SQL/MP software of version 300 or later, NonStop SQL/MP assigns the new catalog the version of the SQL software running on the node on which the catalog resides. You can change the version with `UPGRADE CATALOG` or `DOWNGRADE CATALOG`, but no catalog can have a newer version than the software on the catalog's node, or an older version than that of the newest-version object registered in the catalog.

A system catalog follows the same rules as any other catalog except that you use `UPGRADE SYSTEM CATALOG` and `DOWNGRADE SYSTEM CATALOG` to change the version. A system catalog can also register user catalogs of newer versions (because these are not considered SQL/MP objects) but cannot register objects of newer versions.

The command `GET VERSION OF CATALOG` returns the version of a catalog.

Object Versions

Each SQL/MP object (table, view, index, constraint, or collation) has a version that indicates the oldest version of NonStop SQL/MP that can support that object.

The version of an object depends on the features used in the object and in other objects on which that object depends. For example, if you create a table that uses only features that were available in version 1 of NonStop SQL/MP, the table is associated with version 1, even if you create it with version 315 software. If you add a column to

the table that has the NCHAR data type, however, the version of the table becomes version 300 (because the NCHAR data type is first supported in version 300).

Because the version of an object affects the version of any object that depends upon that object, changing the version of an object can automatically change the version of other objects that depend on that object. For example, adding an index that has a newer version than the associated table changes the version of the table. If the table has a dependent view, the operation also changes the version of the view.

The GET VERSION command returns the version of a table, view, index, or collation.

Program Versions

Each compiled SQL/MP program has three different versions associated with it:

- The host object SQL version (HOSV) is the version of the host language compiler that compiled the program. It indicates the oldest version of the SQL compiler that can compile the program.
- The program catalog version (PCV) is the version of the newest-version SQL feature used in the program. It indicates the oldest version of a catalog that can register the program. A program can access objects with versions older or newer than the PCV of the program, but a program cannot be registered in a catalog that has a version older than the PCV of the program.
- The program format version (PFV) is the version of the SQL compiler that compiled the program. It indicates the oldest version of SQL/MP software that can execute the program and the newest version of objects that the program can access. A program can be registered in a catalog that has a version older or newer than the PFV of the program, but a program cannot access an object that has a version newer than the PFV of the program.

A host language compiler cannot have a newer version than the SQL/MP software on the same node, but can have an older version.

Host language compiler versions

Each host language compiler that supports SQL/MP has a version that indicates the oldest version of SQL/MP software that can SQL-compile object programs produced by the host language compiler. The version of the host language compiler becomes the host object SQL version (HOSV) of the object programs it produces.

The GET VERSION OF PROGRAM command returns the HOSV, PCV, or PFV of a program.

VERSIONS Table

The VERSIONS table is a catalog table that stores version information about the catalog. The version information is also replicated in the SQL.CATALOGS table. [Table V-1](#) describes the contents of the VERSIONS table.

Table V-1. The VERSIONS Table

Column Name	Data Type	Description
1 SUBSYSTEMNAME *	CHAR(30)	Name of subsystem where catalog resides
2 VERSION	CHAR(4)	Version of catalog: A010=1, A011=2, ...A315=315, and so forth
3 VERSIONUPGRADETIME	LARGEIN SIGNED	Julian timestamp for last upgrade or downgrade of catalog
4 CATALOGCLASS	CHAR(1)	S if system catalog U if user catalog
5 CATALOGVERSION	SMALLINTUNSIGNED	Version number of catalog
6 CATALOGFORMAT	SMALLINTUNSIGNED	Format number of catalog (version number of oldest software that can read or write the catalog)

* Indicates primary key

The columns SUBSYSTEMNAME through CATALOGCLASS (1 through 4) were created in version 1. The columns CATALOGVERSION and CATALOGFORMAT (5 through 6) were added in version 300.

Views

A view is a logical table created with the CREATE VIEW statement and derived by projecting a subset of columns, restricting a subset of rows, or both, from one or more base tables or other views. A view has a file label but has no actual data separate from the data in the tables on which it is defined.

A view name must be a Guardian name. The fully expanded name of the view must be unique among object names in the network.

A view is either a protection view or a shorthand view:

- A protection view is derived from a single table and can be read, updated, and secured.
- A shorthand view is derived from one or more tables or other views. A shorthand view can be read but cannot be updated. A shorthand view can be secured for purge authority, but any user who has read access to the tables and views underlying the view can read the view.

Protection views are always valid. A shorthand view becomes invalid if a user who lacks purge authority for the view itself purges any of the tables or views underlying the view. To make an invalid shorthand view valid, the owner of the invalid view must purge and re-create the view.

VIEWS Table

The VIEWS table is a catalog table that describes the views available for the base tables. [Table V-2](#) describes the contents of the VIEWS table.

Table V-2. The VIEWS Table

Column Name	Data Type	Description
1 VIEWNAME *	CHAR (34)	Name of view
2 PROTECTION	CHAR (1)	Y if protection view N if shorthand view
3 VALIDDEF	CHAR (1)	Y if definition valid N if definition not valid
4 AUDIT	CHAR (1)	Y if all underlying tables are audited N if all underlying tables are nonaudited M if based both on audited and nonaudited tables
5 WITHCHECKOPTION	CHAR (1)	Y if defined with WITH CHECK OPTION N if not
6 INSERTABLE	CHAR (1)	Y if row inserts allowed N if not
7 VIEWTEXT	VARCHAR (3000)	Text of statement used to define view

* Indicates primary key

The VIEWS table was created in version 1, and subsequent modifications have not been made.

Guardian names in the VIEWS table (including names of tables and views in the statements stored in the VIEWTEXT column) are fully qualified. Table and view names in the VIEWTEXT column are actual file names, never DEFINE names. SQL replaces any DEFINE names with actual file names at the time you create the view.

All CHAR and VARCHAR columns in the VIEWS table use uppercase characters except for lowercase string literals specified as part of the search condition in the VIEWTEXT column. Search conditions that specify the system default multibyte character set are stored as if you specified the actual character set. (For example, if the system default multibyte character set is Kanji, the literal N"" is stored as _KANJI"....".)

VOLUME Command

VOLUME is an SQLCI command that changes the current node, volume, or subvolume defaults for the SQLCI session. VOLUME also sets the VOLUME attribute of the `=_DEFAULTS DEFINE`.

Defaults you set with VOLUME remain in effect until you exit SQLCI or until you enter a `SYSTEM`, `VOLUME`, or `ALTER DEFINE` that changes the defaults.

```
VOLUME [ \node ]
        [ [ \node. ] [ $volume ] ] ;
        [ [ \node. ] [ $volume. ] subvol ]
```

`\node`

specifies the node to be the current default node.

`volume`

specifies the volume to be the current default volume.

`subvol`

specifies the subvolume to be the current default subvolume.

Considerations—VOLUME

- If you enter VOLUME with no parameters, SQLCI resets the current default node, volume, and subvolume to their values at the start of the SQLCI session. However, if you specify at least one parameter but omit one or more, SQLCI changes only what you specify and leaves other values unchanged.
- Changing the current defaults for file name expansion does not affect names in prepared statements unless a `CONTROL QUERY BIND NAMES AT EXECUTION` directive was in effect at the time a `PREPARE` compiled the statement. If you want new defaults to take effect in other prepared statements, you must use `PREPARE` to recompile the statements. For information, see [CONTROL QUERY Directive](#) on page C-74 or [Name Resolution](#) on page N-2.

Examples—VOLUME

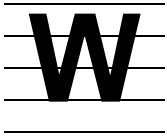
- This example changes the current default system to `\SYS1`:

```
>> VOLUME \SYS1;
```
- This example changes the default volume and subvolume to `$VOL1.INVENT`:

```
>> VOLUME $VOL1.INVENT;
```

- This example resets the default node, volume, and subvolume to their values at the start of the SQLCI session:

```
>> VOLUME ;
```

WHENEVER DIRECTIVE

WHENEVER is a host program directive that specifies an action to take when an error, warning, or no-rows-found condition occurs.

WHENEVER	{	NOT FOUND	}	[CONTINUE]
				[GOTO : <i>host-id</i>]
				[GO TO : <i>host-id</i>]
				[CALL : <i>host-id</i>]
				[PERFORM : <i>host-id</i>]

```
{ NOT FOUND }  
{ SQLERROR }  
{ SQLWARNING }
```

specifies a condition to test for:

NOT FOUND	A no-rows-found condition (SQLCODE 100)
SQLERROR	An error (a negative SQLCODE value)
SQLWARNING	A warning (a positive SQLCODE value other than 100)

SQL tests for the condition after each DCL, DDL, and DML statement for which the WHENEVER directive is in effect. (To end testing, specify WHENEVER with the same condition, but no action.)

In a SELECT through a cursor, NOT FOUND means no rows or all rows qualify. In statements with a WHERE clause, NOT FOUND means no rows satisfy the WHERE clause. In a FETCH after a series of fetches, NOT FOUND means all rows were fetched.

```
[ CONTINUE ]  
[ GOTO :host-id ]  
[ GO TO :host-id ]  
[ CALL :host-id ]  
[ PERFORM :host-id ]
```

specifies the action to take:

CONTINUE	continue with next statement
GOTO : <i>host-id</i>	pass control to location <i>host-id</i>
GO TO : <i>host-id</i>	pass control to location <i>host-id</i>
CALL : <i>host-id</i>	execute <i>host-id</i> (Not COBOL85)
PERFORM : <i>host-id</i>	execute <i>host-id</i> (COBOL85 only)

host-id is an identifier that specifies a location in the host language program. For more information, see the SQL/MP programming manual for your host language.

If you do not specify an action, SQL discontinues checking for the specified condition.

Consideration—WHENEVER Directive

The WHENEVER directive applies to source lines of a program that are sequentially compiled (including text brought in by SOURCE, COPY, and INLINE directives) The directive stays in effect until SQL detects another WHENEVER directive for the same condition.

WHERE CLAUSE

WHERE clauses on SELECT, DELETE, or UPDATE statements are search condition clauses that specify criteria for choosing rows from tables or views. For more information, see [SELECT Statement](#) on page S-18, [DELETE Statement](#) on page D-39, or [UPDATE Statement](#) on page U-3.

WHERE clauses on SQLCI utility commands qualify the set of files on which the command is to operate.

For examples of the WHERE clause, see [Qualified Fileset List](#) on page Q-1 and topics for other language elements related to the WHERE clause, such as [SELECT Statement](#) on page S-18, [DELETE Statement](#) on page D-39, [UPDATE Statement](#) on page U-3, and [EXISTS Predicate](#) on page E-12.

WINDOW OPTION

WINDOW is an option of the SQLCI report writer SET LAYOUT command that specifies a print item or print position to display at the left edge of the output device. It enables you to display the portion of a report that extends beyond the right edge of the output device.

<pre>WINDOW { TAB <i>number</i> } { <i>column</i> }</pre>
--

TAB number

specifies an integer in the range 1 through 255 that is the print position of the output line to display at the left edge of the output device. (The first position in the output line is 1, and each print position occupies one byte.) The default is TAB 1.

You can use this clause at the select-in-progress prompt (S>) or at the standard SQLCI prompt (>>).

column

identifies a print item in the detail print list to begin printing at the left edge of the output device. *column* can be a column name, an alias, or COL *number* (which

specifies the position of the column in the select list). *column* cannot be a detail alias.

You can use this clause only at the select-in-progress prompt (S>).

If you specify a column that is not in the detail print list, the report writer ignores the WINDOW command. If the column appears in the detail print list more than once, its first occurrence is the one that prints at the left edge of the output device.

Consideration—WINDOW

If WINDOW is set to a column identifier (as opposed to a TAB position) when a SELECT command terminates, the report writer resets WINDOW to TAB 1.

Examples—WINDOW

- This example defines a report format for the rows specified in the SELECT command at the beginning of the example. The WINDOW option of the SET LAYOUT command specifies that only the rightmost portion of the report (beginning at print position 40) should be displayed when the LIST command executes.

```
>> SELECT EMPNUM, LAST_NAME, SALARY, SALARY * .05
+> FROM PERSNL.EMPLOYEE;
S> NAME COL 4 BONUS;
S> DETAIL "*", EMPNUM, LAST_NAME, SALARY, SALARY * 1.1 NAME
+> NEW_SALARY, BONUS;
S> SET LAYOUT WINDOW TAB 40;
S> LIST;
```

- These SET LAYOUT commands use the WINDOW option to specify different vertical segments of the report defined in the previous example. (You could use any one in place of the previous SET LAYOUT command.) As in that example, these must be preceded by an appropriate SELECT command and followed by a LIST command to actually print a report.

```
S> SET LAYOUT WINDOW SALARY;   Displays SALARY, NEW_SALARY, and
                                BONUS
S> SET LAYOUT WINDOW COL 2;    Displays LAST_NAME, SALARY,
                                NEW_SALARY, and BONUS
S> SET LAYOUT WINDOW TAB 44;   Displays BONUS (print position 44)
```

WITH SHARED ACCESS OPTION

WITH SHARED ACCESS is an option available on some DDL statements that specifies that the DDL operation is to allow concurrent read-write DML access and read-only utility access to the objects on which it operates during all but the final phase of the operation.

DDL statements that include the WITH SHARED ACCESS option initiate potentially long-running operations that prohibit concurrent INSERT, DELETE, UPDATE, and utility operations unless you explicitly specify WITH SHARED ACCESS.

	[NAME <i>operation-name</i>]
	[]
	[[TO <i>collector</i>]]
WITH SHARED ACCESS	[REPORT [ON]
	[[OFF]
	[]
	[{ COMMIT [WORK] <i>commit-options</i> }]
	[{ ROLLBACK [WORK]]

NAME *operation-name*

specifies an SQL identifier as the name of the operation.

If you omit the NAME option, the name of the operation is the first two words of the statement that initiated the operation concatenated by an underscore (for example, CREATE_INDEX).

For more information, see [NAME Option](#) on page N-2.

REPORT [TO *collector* | ON | OFF]

controls EMS reporting for the operation.

Omitting the option entirely is equivalent to specifying REPORT ON and sends event messages for the operation to \$0, the default EMS collector. Specifying REPORT without an option is equivalent to specifying REPORT OFF and suppresses event messages for the operation.

For more information, see [REPORT Option](#) on page R-3.

{ COMMIT [WORK] *commit-options* }

{ ROLLBACK [WORK] }

specifies the start time, the timeout period for lock requests, and the handling of retryable errors for the commit phase of the operation. The default is,

COMMIT WHEN READY
TIMEOUT DEFAULT ONCOMMITERROR ROLLBACK WORK

For more information, see [COMMIT Option](#) on page C-52.

Considerations—WITH SHARED ACCESS

- Restrictions

- You cannot use WITH SHARED ACCESS on a statement that executes within a user-defined TMF transaction.
- You cannot use WITH SHARED ACCESS unless each source object (for example, the table being indexed by CREATE INDEX or the partition being moved by ALTER INDEX or ALTER TABLE) in the operation is audited.
- You cannot use WITH SHARED ACCESS for an ALTER TABLE or ALTER INDEX statement unless each source object and each target object (for example, the new location of a partition being moved by ALTER INDEX or ALTER TABLE) in the operation resides on a node running version 315 or later of NonStop SQL/MP.
- You cannot use WITH SHARED ACCESS on a statement that performs cross-node operations, where the TMF on one node provides Format 2 audit trail file support, and the TMF on the other node provides only Format 1 audit trail file support.

- TMF audit trail requirements

An operation that uses WITH SHARED ACCESS cannot complete successfully unless the TMF audit trail generated during the operation is available for reading later in the operation. If a required audit trail has been overwritten, a WITH SHARED ACCESS operation cancels changes made to the database and terminates.

When performed on a source object that has a valid TMF online dump, an operation that uses WITH SHARED ACCESS generates audit information for the target object. The target might not audit to the same audit trail as the source.

Lengthy operations that use WITH SHARED ACCESS might require an operator to mount tapes of previously taken TMF audit dumps. Requests to mount TMF audit dump tapes for WITH SHARED ACCESS operations are not distinguishable from other requests to mount TMF audit dump tapes. Such requests are generally sent to an operator's console. SQL does not return information about such requests to the terminal or process that started the operation.

- Phases of a WITH SHARED ACCESS operation

An operation specified by a statement that includes the WITH SHARED ACCESS option occurs in these phases:

1. Initialization and load phase

SQL reads catalog entries for existing objects involved in the operation (the “source objects”) and creates any new objects for the operation with auditing disabled. For CREATE INDEX operations, SQL also sets the NOAUDITCOMPRESS file attribute for the table being indexed unless it is already set. All this activity occurs in one transaction.

Note. Some operations involve one source object and one target object, while others involve many. For example, a simple move of a partition involves one source object and one target object. However, if you create an index on a partitioned table, each partition of the table is a source object for the operation; if the new index is itself partitioned, each partition of the index is a target object for the operation.

SQL then starts an audit fix-up process on each node that contains one or more source objects for the operation and begins copying data from source objects to target objects as needed. For CREATE INDEX, SQL transforms the data as needed. All this activity occurs outside of a transaction. If EMS reporting is turned on, the audit fix-up processes issue status messages throughout the load.

SQL starts each audit fix-up process as a named process running under the process access ID of the user that started the WITH SHARED ACCESS operation. However, audit fix-up processes switch to the super ID during portions of their execution, then switch back to the initial user ID. (The audit fix-up processes handle this change automatically, but you might notice that these processes sometimes appear in lists of processes running under your user ID and sometimes do not.)

After the load finishes, SQL executes a transaction that enables auditing, if it is not already enabled, on the target objects, after which, you can take online dumps of the target objects if necessary. (Online dumps are not necessary for merge and move boundary requests.) If your operation uses EMS reporting, an EMS message is issued.

2. Audit fix-up phase

The audit fix-up processes search TMF audit trails to find audit information for the source objects and update the target objects to reflect any changes made since the load of the corresponding records. When the updates are complete, SQL is ready to commit the operation.

Although the target objects are audited, the changes made by the audit fix-up processes during this phase do not occur within TMF transactions unless the source objects have file-recovery protection (that is, if valid online dumps exist for the source objects).

Depending on the COMMIT option in effect for the operation, SQL either moves into the commit phase (the default), waits for the appropriate time window in which to move into the commit phase, rolls back the operation and terminates with an error because the time window has passed, or issues

warnings 1618 and 1619 to notify the user that the operation is ready to commit and waits for the user to respond with a CONTINUE statement.

If the operation cannot move into the commit phase immediately (because it must wait for a time window or a CONTINUE statement) the audit fix-up processes continue reading audit trails and updating target objects to maintain the ready-to-commit state.

3. Commit phase and command completion

SQL begins a transaction and acquires an exclusive table lock on each source object. This stops DML transaction activity against the source objects so that the audit fix-up processes can complete their work.

When the audit fix-up processes have finished, SQL acquires label and file locks on all objects that participate in the operation, except source objects, which are already locked.

(For ALTER INDEX MOVE or ALTER TABLE MOVE, objects that participate in the operation include all partitions of the index or table involved in the move, not just the specific partition being moved.)

SQL then updates file labels and catalog tables. At this time, the commit phase has completed. A few extra steps such as program invalidation are performed within the same transaction. After these final steps are performed the transaction commits and the operation completes.

- Error considerations

Errors before or after the commit phase of a WITH SHARED ACCESS operation cause SQL to cancel changes to the database and terminate the operation, as do nonretryable errors during the commit phase.

Retryable errors during the commit phase cause SQL to take the action specified in the ONCOMMITERROR option of the COMMIT specification in effect. The default is to cancel changes to the database and terminate the operation. For more information, see COMMIT OPTION.

If the process that started a WITH SHARED ACCESS operation terminates abnormally, the DDL operation in progress stops without being either committed or canceled. (This halting also occurs if the user fails to issue a CONTINUE statement in response to warning 1619, as discussed under CONTINUE.) If this event occurs, the changes are not made to the database, but you (or another user with the super ID) must use CLEANUP to remove the new objects. If the operation was a CREATE INDEX operation on a table with the AUDITCOMPRESS attribute, you must also use ALTER TABLE to reset the AUDITCOMPRESS attribute.

- Performance considerations

Operations that use WITH SHARED ACCESS usually require more time to finish than those that do not. However, because WITH SHARED ACCESS operations allow concurrent read and write access to the source partition, such operations

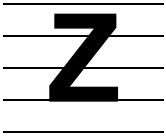
cause far less application downtime than equivalent operations without WITH SHARED ACCESS.

The duration of a WITH SHARED ACCESS operation increases with the number and length of transactions on the node that contains the source partition, particularly with the number and length of transactions that involve the source partition and the amount of activity on the audit trail used for the source partition.

Example—WITH SHARED ACCESS

The command CREATE INDEX uses the WITH SHARED ACCESS option:

```
CREATE INDEX EMPLOYEE2
  ON EMPLOYEE (JOBCODE) CATALOG PERSONL
  WITH SHARED ACCESS NAME CR_IND_EMP2
  COMMIT WHEN READY TIMEOUT NEVER;
```

! Command

! (exclamation point) is an SQLCI command that re-executes a statement or command without modifying a previous statement or command in the history buffer or a current report formatting command. For more information, see [HISTORY Command](#) on page H-5.

```
! [ text ] [ ; ]  
  [ [-]number ]
```

text

specifies the most recent version of a command in the history buffer or a stored report formatting command. The command must begin with *text*, but *text* need only be as many characters as necessary to identify the command.

number

is an integer that specifies a command in the history buffer.

If *number* is negative, it indicates the position of the command in the history buffer relative to the current command; if *number* is positive, it is the ordinal number of a command in the history buffer.

To re-execute the previous command, you can enter an exclamation point (!) without specifying text or a number. If you enter more than one SQLCI command on a line, the exclamation point re-executes only the last command on the line.

Examples—!

- To reexecute the last SELECT command, enter:

```
>> ! SELECT
```
- To reexecute the second to the last command entered, enter:

```
>> !-2
```

=_AUDSERV_XSWAP_node DEFINE

=_AUDSERV_XSWAP_node is a system DEFINE that specifies a swap volume for the audit fix-up process on the specified node.

<pre>ADD DEFINE =_AUDSERV_XSWAP_node, CLASS MAP, FILE volume</pre>
--

node

is the name of the node (without the usual leading “\”) for which the swap volume of the audit fix-up process is to be set.

volume

is the name of the swap volume for the audit fix-up process on the node.

Consideration—=_AUDSERV_XSWAP_node

An operation that uses the WITH SHARED ACCESS option starts an audit fix-up process on each node in the network that contains at least one source object used in the operation. (There might be more than one node for CREATE INDEX operations on network-partitioned tables.) The default swap volume for an audit fix-up process is the volume that contains the program file for the process.

Because an audit fix-up process generates a large swap file, you might want to specify an alternate swap volume for such a process with the =_AUDSERV_XSWAP_node DEFINE.

Example—=_AUDSERV_XSWAP_node

- These SQLCI commands specify swap volumes for audit fix-up processes on nodes \REG1 and \REG2:

```
>>ADD DEFINE =_AUDSERV_XSWAP_REG1, CLASS MAP, FILE
\REG1.$SCR;
>>ADD DEFINE =_AUDSERV_XSWAP_REG2, CLASS MAP, FILE
\REG2.$VM3;
```

=_DEFAULTS DEFINE

=_DEFAULTS is a system DEFINE that specifies the current default node, volume, subvolume, and catalog. =_DEFAULTS determines how to expand partially qualified Guardian names.

=_DEFAULT always has a VOLUME attribute that specifies the current default volume and subvolume. If =_DEFAULT has a CATALOG attribute, that attribute specifies the current default SQL catalog; if not, the VOLUME attribute specifies the subvolume that is the current default SQL catalog.

You cannot rename or delete the =_DEFAULTS DEFINE, but you can display and alter it.

```
ALTER DEFINE =_DEFAULTS,
{
  CATALOG [\node.][$volume.]subvolume
  VOLUME [\node.][$volume.]subvolume
}
```

CATALOG [\node.][\$volume.]subvolume

sets the current default catalog, which is used wherever a catalog name is required but no CATALOG clause is supplied, such as in DDL statements or in the SQL compiler command.

VOLUME [\node.][\$volume.]subvolume

sets the current default node, volume, and subvolume. The file system uses the current defaults to expand a partially specified Guardian name to a fully qualified name.

Considerations—=_DEFAULTS

- Processes started during a TACL session automatically inherit the =_DEFAULTS DEFINE from the TACL session, regardless of the DEFMODE setting.
- You can alter =_DEFAULTS explicitly using the ALTER DEFINE command. You can alter it implicitly using the VOLUME, SYSTEM, and CATALOG commands.

If you alter =_DEFAULTS from SQLCI, you alter it only for the duration of the SQLCI session, not for the TACL session that started the SQLCI session.

- With the introduction of the kernel-managed swap facility, the SWAP option is ignored but is stored so that programs can continue to use the information when creating temporary files.

Examples—=_DEFAULTS

- These commands are all legal commands that alter the =_DEFAULTS DEFINE:

```
ALTER DEFINE =_DEFAULTS, VOLUME \SYS1.$VOL1.PERSNL;
ALTER DEFINE =_DEFAULTS, CATALOG $VOL1.CAT1;
ALTER DEFINE =_DEFAULTS, VOLUME $VOL1.PAYROLL;
VOLUME $VOL1.ORDENTRY;
SYSTEM \SYS2;
ALTER DEFINE =_DEFAULTS, SWAP $VOL1;
```

- This example uses the INFO DEFINE command to display the current setting of the =_DEFAULTS DEFINE:

```
>>INFO DEFINE =_DEFAULTS, DETAIL;
CLASS          DEFAULTS
VOLUME         \SYS2.$VOL1.ORDENTRY
CATALOG        \SYS1.$VOL1.CAT1
SWAP           \SYS2.$VOL1
```

=_SORT_DEFAULTS DEFINE

=_SORT_DEFAULTS is a system DEFINE that specifies defaults for FastSort operations. It can affect SQL performance because SQL uses FastSort for queries and utility operations.

=_SORT_DEFAULTS affects sorts performed as part of an SQL statement executed in parallel if you do not specify scratch and swap files in a configuration file or by some other mechanism. In this case, SQL uses the scratch and swap files specified in the =_SORT_DEFAULTS DEFINE

```
ADD DEFINE =_SORT_DEFAULTS,

    CLASS SORT [ , param value ] ...

param value is:

    { BLOCK block-size
      CPU cpu-number
      CPUS subsort-cpu-list
      MODE mode-type
      NOSCRATCHON ( volume-list )
      NOTCPUS cpu-list-not-subsort
      PRI process-priority
      PROGRAM file
      SCRATCH file
      SCRATCHON ( volume-list )
      SEGMENT extended-segment-size
      SUBSORTS define-list
      SWAP file-name
      VLM { ON | OFF }
    }
```

param value

specifies a FastSort parameter and a value for that parameter. FastSort parameters commonly used with NonStop SQL/MP are:

<i>SCRATCH file</i>	specifies the Guardian name of a scratch file or volume to store runs of records sorted by each SORTPROG process. If you specify only a volume name, FastSort automatically creates the scratch file; this method is recommended for parallel sort operations.
<i>SWAP file</i>	specifies the Guardian name of a swap file or volume for the extended memory segment. If you specify only a volume name, FastSort automatically creates the swap file; this method is recommended for parallel sort operations.
<i>VLM { ON OFF }</i>	specifies use of additional memory for sorting; improves performance in certain situations (usually for LOAD, but not for CREATE INDEX). Do not use VLM for parallel sort operations.

For information about other FastSort parameters and the parameters just described, see the *FastSort Manual*.

Considerations—=_SORT_DEFAULTS

- By using =_SORT_DEFAULTS, you can improve performance for DML operations by giving the FastSort process sufficient space for sorting on an alternate disk volume. Using =_SORT_DEFAULTS can also prevent errors caused when DML operations encounter a full disk.
- If the swap and scratch files you specify do not exist at the time of the FastSort operation, FastSort creates them and uses MAXEXTENTS 160. If you create the files yourself, you must choose an appropriate file size.
- If you specify swap or scratch files on a LOAD command or in the configuration file for parallel execution of a CREATE INDEX or LOAD command, those file specifications override any file specifications in =_SORT_DEFAULTS for that LOAD operation.
- If SMF is installed on your node, you can specify either a virtual or physical volume for the SCRATCH parameter. However, only physical volumes are valid values for the SCRATCHON and NOSCRATCHON parameters.

If you specify a virtual volume for SCRATCH, the sort process ignores any overflow scratch volumes you specify in SCRATCHON and NOSCRATCHON. Because FastSort does not automatically create and manage overflow scratch files when you specify a virtual volume for SCRATCH, the virtual volume must have space available for all scratch files for the sort operation.

- An error occurs if you add =_SORT_DEFAULTS without specifying CLASS SORT either explicitly or through the working attribute set.

For more information about =_SORT_DEFAULTS, see the *TACL Reference Manual* or the *FastSort Manual*.

Example—=_SORT_DEFAULTS

This SQLCI example specifies scratch and swap volumes for FastSort operations:

```
>> ADD DEFINE =_SORT_DEFAULTS, CLASS SORT,
+>      SCRATCH $DISKA, SWAP $DISKB;
```

The specified volumes are to be used for SQL queries or utility operations that call FastSort as long as the DEFINE remains in effect.

=_SQL_CAT_HEAP_LIMIT DEFINE

=_SQL_CAT_HEAP_LIMIT is a system DEFINE that specifies the heap space size for the SQLCAT process. Some SQL DDL or utility statements require a large memory heap in the SQLCAT process. The catalog manager currently allocates 4 MB of heap space that can expand to 16 MB. You can use this DEFINE to allocate a larger heap.

```
ADD DEFINE =_SQL_CAT_HEAP_LIMIT, [CLASS MAP, ]
      FILE Mheap-space-size
```

heap-space-size

is the amount of heap space for the SQLCAT process, in megabytes. This value can range from 8 to 1119. SQL does not use the node, volume, and subvolume portions of the file name if supplied.

Considerations—=_SQL_CAT_HEAP_LIMIT

- If there is an error in *heap-space-size*, SQL returns a warning, and SQLCAT runs with the default heap space size.
- The =_SQL_CAT_HEAP_LIMIT DEFINE can be used from TACL, SQLCI, or a user process (for embedded SQL).

Examples—=_SQL_CAT_HEAP_LIMIT

- This TACL command specifies 32 MB of SQLCAT heap space:

```
32> ADD DEFINE =_SQL_CAT_HEAP_LIMIT, CLASS MAP, FILE M32
```

- This TACL command directs SQL to use its default value for heap space size:

```
33> DELETE DEFINE =_SQL_CAT_HEAP_LIMIT
```

=_SQL_CMP_CPUS_node DEFINE

The =_SQL_CMP_CPUS_node DEFINE is a system DEFINE that directs SQL to limit processors used in a parallel query to a specified node.

```
ADD DEFINE =_SQL_CMP_CPUS_node, CLASS MAP, FILE Xhhhh
```

_node

is the node associated with the processors to use for the query. *_node* is optional; the default value is the current node.

Xhhhh

is the letter X (uppercase) followed by up to four hex characters. X has no purpose other than to make the DEFINE syntactically correct. The hex character or characters represent a unique configuration of available processors. For a sample configuration and conversion from binary code to hex characters, see Considerations in this entry.

Considerations—=_SQL_CMP_CPUS_node

- Scope of =_SQL_CMP_CPUS_node

To specify available processors, add this DEFINE before you compile the query. =_SQL_CMP_CPUS_node affects all parallel query plans compiled while the DEFINE is in effect. To specify a different set of available processors for a new parallel query, reset the DEFINE.

=_SQL_CMP_CPUS_node affects only the location of Executor Server Processes (ESPs) for parallel plans. It does not affect the location of the master executor.

=_SQL_CMP_CPUS_node also has no effect upon:

- Locations of sort processes used in the query
- Locations of disk processes used in the query

- Specifying the processor configuration—mapping binary code to hex code

=_SQL_CMP_CPUS_node syntax requires that you express a configuration of processors to use for the parallel query as a string of up to four hex characters.

This example shows how to express the configuration as binary code, then how to map the binary code to hex code. It assumes there are 16 processors on the current node and that processors 0, 1, 2, 3, 8, 10, 12, and 15 are available.

First, assign each processor number a “1” if it is available or a “0” if it is not available:

	CPU #:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary (ON/OFF) value:		1	1	1	1	0	0	0	0	1	0	1	0	1	0	0	1

The processors in this example are shown in groups of four, because each hex character represents four processors:

Hex	Binary	Hex	Binary
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

Using this binary-to-hex conversion chart, the first group of four processors-0, 1, 2, and 3-with the binary code “1111” map to the single hex character “F.” Mapping binary codes for the remaining three groups of processors yields the hex characters “0,” “A,” and “9,” respectively. Therefore, the complete hex character expression for this processor configuration is “F0A9.” The `=_SQL_CMP_CPUS_node` syntax for this configuration is:

```
ADD DEFINE =_SQL_CMP_CPUS, CLASS MAP, FILE XU0A9;
```

For nodes with fewer than 16 processors, SQL assumes that the missing trailing hex characters are zero.

- Availability of processors at compile time

When you compile a parallel query, the compiler assigns ESPs to the processors specified as available in `=_SQL_CMP_CPUS_node`. If a processor is unavailable when you compile the query no ESP is assigned to that processor, even if you specified the processor as available in this DEFINE. In this case, the executor chooses a substitute processor for the stranded ESP. The substitute processor does not have to be specified as available in `=_SQL_CMP_CPUS_node`.

=_SQL_CMP_DOUBLE_SBB_OFF DEFINE

`=_SQL_CMP_DOUBLE_SBB_OFF` is a system DEFINE that directs SQL not to use file system double buffering.

```
ADD DEFINE =_SQL_CMP_DOUBLE_SBB_OFF, CLASS MAP,
FILE filename
```

filename

must be a legal Guardian file name but has no purpose except to make the ADD DEFINE command syntactically correct.

Consideration—=_SQL_CMP_DOUBLE_SBB_OFF

=_SQL_CMP_DOUBLE_SBB_OFF disables file system double buffering for queries that are SQL compiled while this DEFINE is in effect.

For information about how to manage double buffering, see the *SQL/MP Installation and Management Guide*.

=_SQL_CMP_DOUBLE_SBB_ON DEFINE

=_SQL_CMP_DOUBLE_SBB_ON is a system DEFINE that directs SQL to use file system double buffering to scan the inner table in a nested join or key-sequenced merge join.

```
ADD DEFINE =_SQL_CMP_DOUBLE_SBB_ON, CLASS MAP,
      FILE filename
```

filename

must be a legal Guardian file name but has no purpose except to make the ADD DEFINE command syntactically correct.

Considerations—=_SQL_CMP_DOUBLE_SBB_ON

- By default, the SQL optimizer considers using file system double buffering for scans that use VSBB and BROWSE ACCESS except for an inner scan of a nested join or key-sequenced merge join. When the =_SQL_CMP_DOUBLE_SBB_ON DEFINE is present, the optimizer also considers using double buffering for these two types of join operations.
 - If both =_SQL_CMP_DOUBLE_SBB_ON and =_SQL_CMP_DOUBLE_SBB_OFF are present, file-system double buffering is not used.
- ² Double buffering can increase performance, but can also increase the likelihood of PFS memory overflow. For information about how to manage double buffering, see the *SQL/MP Installation and Management Guide*.

=_SQL_CMP_EQ_LIMIT DEFINE

=_SQL_CMP_EQ_LIMIT is a system DEFINE that specifies the number of expressions in an equivalence class for which the query rewrite feature adds equality predicates.

```
ADD DEFINE =_SQL_CMP_EQ_LIMIT, CLASS MAP, FILE xnn
```

xnn

is a string containing a single arbitrary alphabetic character followed by one or two digits that represent a decimal value between 0 and 99. If you do not use this DEFINE, SQL uses a default value of 5.

Consideration—=_SQL_CMP_EQ_LIMIT

The optimizer evaluates join plans for different combinations of tables. If the join involves numerous tables, this selection process can be inefficient. When choosing a value for =_SQL_CMP_EQ_LIMIT, set the DEFINE low enough to obtain a reasonable SQL compilation time but high enough to obtain the benefits of the optimization process. Typical values are:

- 0 or 1 SQL does not generate any additional equivalent predicates
- 2 or 3 Increased compile time is negligible
- 4 to 6 Some increased compile time but a wider range of table combinations, allowing a more efficient query plan

Examples—=_SQL_CMP_EQ_LIMIT

- This TACL command specifies a limit of four equivalent predicates for join order and index selection:

```
32> ADD DEFINE =_SQL_CMP_EQ_LIMIT, CLASS MAP, FILE T4
```

- This TACL command directs SQL to use its default value for the number of equivalent predicates:

```
34> DELETE DEFINE =_SQL_CMP_EQ_LIMIT
```

=_SQL_CMP_EVENT DEFINE

=_SQL_CMP_EVENT is a system DEFINE that directs the SQL compiler to log compiler event messages to a file or to the home terminal.

```
ADD DEFINE =_SQL_CMP_EVENT, CLASS MAP, FILE filename
```

filename

must be a legal Guardian file name.

If the named file is an existing, entry-sequenced file, the compiler sends compiler event messages to the file. If the file does not exist or is not an entry-sequenced file, the compiler sends only messages for compiler events of type STMT and PROG and sends the messages to the home terminal.

Format of SQL Compiler Event Messages

SQL compiler event messages can be up to 102 characters long.

Log files with shorter record lengths receive truncated messages.

The event messages contain these fields:

```
86 SQLCOMP type datetime file PID=pid E=err FS=fs-err
```

<i>type</i>	Type of event that occurred
<i>datetime</i>	Date and time of the compilation
<i>file</i>	Name of the program file that was compiled
<i>pid</i>	Process that started the compiler (a process name, if any, or a processor and PIN)
<i>err</i>	SQL error that caused the automatic recompilation. (Appears only on messages of type PROG or STMT.)
<i>fs-err</i>	File-system error that caused the automatic recompilation. (Appears only on messages of type PROG or STMT.)

The types of events are;

BTCH	Recompilation and registration of a program by RESTORE
PREP	Compilation of one statement by a run-time PREPARE statement or an EXECUTE IMMEDIATE command from SQLCI
PROG	Automatic recompilation of a program at run time
STAT	Explicit compilation
STMT	Automatic recompilation of one statement at run time

Considerations—=_SQL_CMP_EVENT

- If you use =_SQL_CMP_EVENT to direct compiler event messages regarding automatic recompilations (PROG and STMT type events) to a home terminal, the home terminal used is the home terminal for the process that invoked the SQL compiler. This selection is normally the home terminal for the executing TACL for SQLCI session, but you can specify a different home terminal for the process by specifying the TERM option in the RUN command that executes the process.

You cannot direct other types of event messages from the SQL compiler to the home terminal, and you cannot direct event messages regarding automatic compilations to a different terminal or to both a file and the home terminal.

- You cannot globally direct SQL compiler event messages from other processes on a node to a file or home terminal; each affected TACL or SQLCI session must have its own =_SQL_CMP_EVENT DEFINE in effect, although multiple users can direct messages to the same file.

Examples—=_SQL_CMP_EVENT

- This example creates a file called LOGFILE on volume \$X subvolume Y and directs the SQL compiler to send event messages there, as shown:

```
30> FUP CREATE LOGFILE, TYPE E, REC 102, NO AUDIT;
31> ADD DEFINE =_SQL_CMP_EVENT, CLASS MAP, FILE $X.Y.LOGFILE;
```

The DEFINE affects only SQL compiler messages from a TACL process in which the DEFINE is in effect. After the file is created, however, users of other TACL processes can direct compiler messages to the same file by adding a similar DEFINE.

- This TACL command directs the SQL compiler to log PROG and STMT events to the user's terminal by specifying a nonexistent file for the =_SQL_CMP_EVENT DEFINE:

```
32> ADD DEFINE =_SQL_CMP_EVENT, CLASS MAP, FILE NOFILE;
```

=_SQL_CMP_EVENT_NO0 DEFINE

=_SQL_CMP_EVENT_NO0 is a system DEFINE that suppresses event messages that the SQL compiler normally sends to \$0 when automatic recompilations occur.

```
ADD DEFINE =_SQL_CMP_EVENT_NO0, CLASS MAP, FILE filename
```

filename

must be a legal Guardian file name but has no purpose except to make the ADD DEFINE command syntactically correct.

Default Event Messages

If the `=_SQL_CMP_EVENT_NO0` DEFINE does not exist, the SQL compiler issues a message to \$0 whenever an automatic recompilation occurs.

The event messages contain these fields:

```
86 SQLCOMP type datetime file PID=pid E=err FS=fs-err
```

type PROG (indicating automatic recompilation of an entire program) or
STMT (indicating automatic of a single statement)

datetime the date and time of the compilation

file the name of the program file

pid identifies the process that started the compiler (a process name, if
any, or a processor and PIN)

err the SQL error that caused the automatic recompilation

fs-err the file-system error that caused the automatic recompilation

Compiler event messages displayed on the console have an EMS subsystem ID rather than an SQL ID, because SQL compiler event messages are not EMS messages. The compiler sends the messages to \$0, where they are converted to EMS messages and given an EMS subsystem ID.

Consideration—=_SQL_CMP_EVENT_NO0

`=_SQL_CMP_EVENT_NO0` affects only event messages issued within the TACL or SQLCI session in which the DEFINE is in effect.

You cannot globally disable the logging of such messages to \$0 except to have each TACL process on the node add the `=_SQL_CMP_EVENT_NO0` DEFINE.

Example—=_SQL_CMP_EVENT_NO0

This TACL command directs the SQL compiler not to send event messages regarding automatic recompilations from the TACL or SQLCI session to \$0:

```
32> ADD DEFINE =_SQL_CMP_EVENT_NO0, CLASS MAP, FILE NOFILE
```

=_SQL_CMP_NO_KS_MJOIN DEFINE

=_SQL_CMP_NO_KS_MJOIN is a system DEFINE that inhibits SQL from using the key-sequenced merge join operation.

```
ADD DEFINE =_SQL_CMP_NO_KS_MJOIN, CLASS MAP, FILE name
```

name

must be a legal Guardian file name but has no purpose except to make the ADD DEFINE command syntactically correct. If you do not set this DEFINE, a key-sequenced merge join is considered.

Examples—=_SQL_CMP_NO_KS_MJOIN

- This TACL command directs SQL to avoid using KSMJ:

```
32> ADD DEFINE =_SQL_CMP_NO_KS_MJOIN, CLASS MAP, FILE T1
```

- To direct SQL to resume using KSMJ, delete the DEFINE as shown:

```
34> DELETE DEFINE =_SQL_CMP_NO_KS_MJOIN
```

=_SQL_*cmp_node* DEFINE

=_SQL_*cmp_node* is a set of system DEFINES that identify files that hold components of SQL/MP software. Only a user logged on with an ID from the group 255 can use these DEFINES to direct SQL to use alternate versions of its normal components.

```
ADD DEFINE =_SQL_cmp_node, CLASS MAP, FILE name
```

cmp

is one of these components:

<i>cmp</i>	Normal \$SYSTEM.SYSTEM file for component
AUD	AUDSERV
CAT	SQLCAT
CI2	SQLCI2
CMP	SQLCOMP
UTL	SQLUTIL

node

is the name of the node (without the usual leading “\”) that uses the specified object file as a component.

name

is the name of the object file to use in place of the standard \$SYSTEM.SYSTEM object file for the component.

Consideration—=_SQL_cmp_node

- You can use a =_SQL_cmp_node DEFINE only if you are logged on as a user in group 255. SQL ignores the DEFINE for other users.
- If the DEFINE =_SQL_CI2_node is in effect during INITIALIZE SQL, the alternate file specified is the file compiled and registered in the system catalog by the INITIALIZE SQL operation.

Specifying an appropriate alternate CI2 file at initialization time is the way you install a licensed SQLCI2.

Example—=_SQL_cmp_node

This TACL command adds two =_SQL_cmp_node DEFINES:

```
37> SET DEFINE CLASS MAP
38> ADD DEFINE =_SQL_CMP_SYS1, FILE
\SYS1.$VOL1.SYSTEM.SQLCOMP
38> ADD DEFINE =_SQL_CI2_SYS1, FILE
\SYS1.$VOL1.SYSTEM.SQLCI2L
```

=_SQL_EXE_DOUBLE_SHUTOFF DEFINE

=_SQL_EXE_DOUBLE_SHUTOFF is a system DEFINE that directs SQL to disable file system double buffering when a specified level of memory use is reached.

```
ADD DEFINE =_SQL_EXE_DOUBLE_SHUTOFF, CLASS MAP, FILE Xn
```

Xn

is the letter X (uppercase) followed by a number from 0 (zero) through 10 that specifies tenths of the process file segment (PFS). It represents the limit above which double buffering is not used. If this limit is reached, double buffering is shut off for that query. No notification is given that this has occurred. If the PFS load decreases, double buffering is turned on again for a new query. When *n* is zero, the executor never uses double buffering.

Considerations—=_SQL_EXE_DOUBLE_SHUTOFF

- When you increase the memory use threshold, you increase the use of double buffering but also increase the likelihood of a PFS memory overflow.
- If the DEFINE is absent, the PFS memory-use threshold defaults to 70 percent.

- For information about how to manage double buffering, see the *SQL/MP Installation and Management Guide*.

=_SQL_EXE_ESPS_CK_CMEN DEFINE

=_SQL_EXE_ESPS_CK_CMEN is a system DEFINE that directs SQL to communicate with the \$CMEN process whenever it must create a new ESP.

ADD DEFINE =_SQL_EXE_ESPS_CK_CMEN, CLASS MAP, FILE <i>name</i>
--

name

must be a legal Guardian file name but has no purpose except to make the ADD DEFINE command syntactically correct.

Consideration—=_SQL_EXE_ESPS_CK_CMEN

In previous RVUs, SQL would send a message to the \$CMEN process before creating an ESP. \$CMEN would reply with a recommendation for a processor in which to create the new ESP and a priority for running the ESP. However, SQL would ignore the recommendations returned by the \$CMEN process. Instead, SQL would create the new ESP in the processor and at the priority determined by the optimizer (and stored in the query execution plan).

With the D43 version of SQL/MP software, SQL does not send a message to the \$CMEN process before creating a new ESP. For parallel queries involving many ESPs, this strategy improves the performance of process creation and reduces query start-up time.

The =_SQL_EXE_ESPS_CK_CMEN DEFINE allows you to direct SQL to notify the \$CMEN process whenever SQL creates a new ESP. However, even when SQL notifies the \$CMEN process before creating an ESP, it does nothing with the information returned by \$CMEN.

Example—=_SQL_EXE_ESPS_CK_CMEN

This example directs SQL to send a message to the \$CMEN process before creating each ESP:

```
30> ADD DEFINE =_SQL_EXE_ESPS_CK_CMEN, CLASS MAP, FILE NOFILE
```


=_SQL_EXE_USE_SWAPVOL DEFINE

=_SQL_EXE_USE_SWAPVOL is a system DEFINE that directs SQL to allocate temporary tables for serial plans on the swap volume for the process.

```
ADD DEFINE =_SQL_EXE_USE_SWAPVOL, CLASS MAP, FILE name
```

name

must be a legal Guardian file name but has no purpose except to make the ADD DEFINE command syntactically correct.

Considerations—=_SQL_EXE_USE_SWAPVOL

- You can use the =_SQL_EXE_USE_SWAPVOL DEFINE to change the location of temporary tables from serial plans that might require more space than available on the volumes where the temporary tables would normally be located, causing file-system error 122.

NonStop SQL/MP normally places temporary tables on the same volume as the outermost table in a join. For more information about temporary table placement, see [Temporary Tables](#) on page T-4.

- If SMF is installed on your node and the swap volume is a virtual volume, SQL places temporary tables on the virtual volume when you specify =_SQL_EXE_USE_SWAPVOL.
- =_SQL_EXE_USE_SWAPVOL has no effect on the location of temporary tables for parallel plans or on the SYNCDEPTH of temporary tables, both of which can also lead to error 122.

Another DEFINE, =_SQL_TM_*node_vol*, allows you to redirect temporary tables from both serial or parallel plans from a specific volume to another specific volume and also allows you to specify SYNCDEPTH 1 instead of SYNCDEPTH 0 for the temporary tables. If both DEFINES exist, =_SQL_TM_*node_vol* overrides the effect of =_SQL_EXE_USE_SWAPVOL for temporary files on the specified system and volume.

- Note that the kernel-managed swap facility manages process swap files.

Examples—=_SQL_EXE_USE_SWAPVOL

- This TACL command directs SQL to allocate temporary tables for serial plans on process swap volumes:

```
32> ADD DEFINE =_SQL_EXE_USE_SWAPVOL, FILE NOFILE
```

- To direct SQL to resume allocating temporary tables at the normal location, delete the DEFINE as shown:

```
34> DELETE DEFINE =_SQL_EXE_USE_SWAPVOL
```

=_SQL_MSG_node DEFINE

=_SQL_MSG_node is a system DEFINE that directs SQL to use an alternate message file. It allows you to specify message files that provide SQL messages in languages other than English.

```
ADD DEFINE =_SQL_MSG_node, CLASS MAP, FILE msg-file
```

node

is the name of the node running the SQL software that is to use the alternate message file. Specify the node name without the leading backslash (\).

msg-file

is the name of the alternate message file.

Considerations—=_SQL_MSG_node

- The SQL message file is a key-sequenced file that contains the text of most of the messages displayed by SQL. SQL retrieves messages from the file as needed.

The standard SQL message file, provided with NonStop SQL/MP, is placed in \$SYSTEM.SYSTEM.SQLMSG when NonStop SQL/MP is installed on a node. All the messages in the file are in English.

The =_SQL_MSG_node DEFINE allows you to designate a different file as the SQL message file for SQL if an appropriate file is available on your network. (HP does not provide alternate SQL message files as a standard part of NonStop SQL/MP. You should check at your site to determine what alternate message files, if any, are available to you.)

When SQL must open the SQL message file, SQL determines whether a =_SQL_MSG_node DEFINE is in effect for the node for the process running SQL. If the DEFINE is in effect, SQL uses the message file specified in the DEFINE. If there is no =_SQL_MSG_node DEFINE in effect, or if the file specified in the DEFINE does not exist, is invalid, or is incompatible with the SQL software that attempted to open it, then SQL opens the message file in \$SYSTEM.SYSTEM.SQLMSG.

- To use an alternate message file from SQLCI, you must specify an =_SQL_MSG_node DEFINE in TACL before you start the SQLCI session. SQL opens the message file at the beginning of an SQLCI session and leaves the file open throughout the session. As a result, changing DEFINES during the session has no effect on the SQL message file used for that session.

If you regularly use an alternate message file from SQLCI, you might want to place an =_SQL_MSG_node DEFINE in your TACLCSTM file so that it is in effect whenever you log on. For information about TACLCSTM, see the *TACL Reference Manual*.

- To use an alternate message file from a program, specify an `=_SQL_MSG_node` DEFINE in TACL before you execute the program, or specify the DEFINE at the beginning of the program. SQL uses the file you specify for all SQL messages from the program.

If you plan to switch message files within a program by changing the `=_SQL_MSG_node` DEFINE in the body of the program, be aware that SQL opens the message file the first time you call any one of these procedures:

- SQLCADISPLAY
- SQLCAFSCODE
- SQLCATOBUFFER
- SQLSADISPLAY
- SQLSA_DISPLAY2
- SQLCA_TOBUFFER2

Whether SQL closes the message file when it returns from the procedure depends on the parameters you supply in the procedure call.

If the message file is left open, SQL uses the same message file the next time you call one of the procedures. If the file is closed, SQL opens the message file (possibly a different one) again the next time you call one of the procedures listed previously.

Each time SQL opens the message file, it selects the file specified in the current `=_SQL_MSG_node` DEFINE, if any. Changes you make to the `=_SQL_MSG_node` DEFINE have no effect on a message file that is already open.

- If SQL attempts to open `$SYSTEM.SYSTEM.SQLMSG` and finds that it does not exist, is invalid, or is incompatible with the SQL/MP software, SQL is unable to issue error, warning, or help messages. Only a few English informational messages that are coded in the software will be available. This situation should never occur, however, if SQL is installed properly and the standard SQL message file is always left in `$SYSTEM.SYSTEM.SQLMSG`.

Examples—=_SQL_MSG_node

- This TACL command designates an alternate message file for SQL/MP software running on system \HQ. (This command would not work if a DEFINE named `=_SQL_MSG_HQ` was already in effect.)

```
ADD DEFINE =_SQL_MSG_HQ, CLASS MAP, FILE \HQ.$SQL.MSG.SPANISH
```

- This TACL command specifies a different SQL message file for node \XYZ if a DEFINE named `=_SQL_MSG_XYZ` already exists.

```
ALTER DEFINE =_SQL_MSG_XYZ, FILE \HQ.$SQL.MSG.FRENCH
```

=_SQL_RECGEN_node DEFINE

=_SQL_RECGEN_node is a system DEFINE that allows a user with super ID authority to specify an alternate location for the FastSort record generator program.

```
ADD DEFINE =_SQL_RECGEN_node, CLASS MAP, FILE prog-file
```

node

is the name of the node (without the usual leading backslash “\”) running the SQL software that is to use the alternate FastSort record generator program.

Example—=_SQL_RECGEN_node

This SQLCI command specifies \$DP1.TEST.RGP as the program file for the FastSort record generator on node \REG1:

```
ADD DEFINE =_SQL_RECGEN_REGS, CLASS MAP, FILE $DP1.TEST.RGP;
```

=_SQL_TM_node_vol DEFINE

=_SQL_TM_node_vol is a system DEFINE that directs NonStop SQL/MP to create temporary tables that would normally go to the specified volume on another specified volume instead. Optionally, you can also use =_SYS_TM_node_vol to specify SYNCDEPTH 1 for the redirected temporary files.

```
ADD DEFINE =_SQL_TM_node_vol, CLASS MAP,
FILE new-loc.syncdepth
```

node

is the name of the node (without the usual leading “\”) from which to redirect temporary files.

vol

is the name of the volume (without the usual leading “\$”) from which temporary files are to be redirected.

new-loc

is the name of the system, volume, and subvolume for the temporary files, specified as a Guardian subvolume name (complete with a “\” preceding the node name and “\$” preceding the volume name).

The subvolume portion of the name is not actually used, but you must include it to make the DEFINE syntactically correct. (Temporary files use names of the form \node.\$vol.#nnnn and do not use subvolume names.)

syncdepth

is the keyword SYNC1 (if you want the temporary files to use SYNCDEPTH 1), or a simple file name (if you want the temporary files to use SYNCDEPTH 0, as usual).

The file name must be a legal Guardian file name, but the file is not used and does not need to exist.

Considerations—=_SQL_TM_node_vol

- You can use the =_SQL_TM_node_vol DEFINE to change the location of temporary tables that might require more space than available in the volumes where the temporary tables would normally be located, causing file-system error 122.

NonStop SQL/MP normally places temporary tables in the same volume as the outermost table in a join. For more information about temporary table placement, see [Temporary Tables](#) on page T-4.

- The disk process normally uses SYNCDEPTH 0 to access temporary SQL/MP tables. This approach saves the overhead of checkpointing operations on the tables to the backup DP2 process but can cause error 122 if a DP2 takeover occurs during execution of a statement.

You can use =_SQL_TM_node_vol to specify SYNCDEPTH 1 for temporary tables if avoiding a potential error 122 justifies the performance penalty for your application.

- =_SQL_TM_node_vol affects temporary tables for both serial and parallel plans.
- If a =_SQL_EXE_USE_SWAPVOL DEFINE exists, =_SQL_TM_node_vol overrides its effect for any temporary files on the specified system and volume.
- If SMF is installed on your node and a query references a table by its logical name, you can use =_SQL_TM_node_vol to:
 - Locate temporary tables on a particular physical volume in the same virtual volume as the table
 - Locate temporary tables on a different virtual volume than the table
- For parallel queries, only physical volumes are candidates for temporary tables. If you specify a virtual volume for =_SQL_TM_node_vol when parallel execution is on, the optimizer ignores this DEFINE.

Examples—=_SQL_TM_node_vol

- This TACL command directs NonStop SQL/MP to create all temporary tables that would normally go to volume \A.\$B on volume \X.\$Y instead:

```
31> ADD DEFINE =_SQL_TM_A_B, CLASS MAP, FILE \X.$Y.Z.NOFILE;
```

- This TACL command directs NonStop SQL/MP to create all temporary tables that would normally go to volume \NY.\$HDQ on volume \NY.\$SCR instead and to use SYNCDEPTH 1 for the tables rather than the usual SYNCDEPTH 0:

```
32> ADD DEFINE =_SQL_TM_NY_HDQ, FILE \NY.$SCR.TMP.SYNC1
```

- Both DEFINES shown in the previous examples can exist simultaneously, directing different sets of temporary tables to different locations. To redirect temporary tables to normal locations, delete the DEFINES with these commands:

```
35> DELETE DEFINE =_SQL_TM_A_B  
36> DELETE DEFINE =_SQL_TM_NY_HDQ
```

Index

Numbers

255, user number [G-7](#), [S-12](#)

A

A (alphanumeric) descriptor [A-61](#)

Abort transaction [R-22](#)

Access options

BROWSE [A-1](#)

concurrency summary [A-3](#)

DDL statements [W-4](#)

description of [A-1](#)

DML statements [A-1](#)

INSERT statement [I-15](#)

REPEATABLE [A-1](#)

STABLE [A-1](#)

UPDATE statement [U-3](#)

Access path

alternate [E-16](#)

controlling [C-79](#), [C-85](#), [C-86](#)

primary [E-16](#)

ACCESS PATH option

CONTROL TABLE directive [C-79](#)

controlling [C-79](#), [C-85](#)

in EXPLAIN report [E-16](#)

Access type, in EXPLAIN report [E-16](#)

Ada language [S-69](#)

ADD COLUMN clause

See ALTER TABLE statement

ADD DEFINE command [A-4](#)

ADD PARTITION clause

ALTER INDEX statement [A-18](#)

ALTER TABLE statement [A-36](#)

Adding partitions

index [A-18](#), [A-25](#)

table [A-36](#)

Aggregate functions

AVG [A-76](#)

COUNT [C-130](#)

description of [A-6](#)

MAX [M-1](#)

MIN [M-3](#)

SUM [S-89](#)

Alias

assigning to column [N-1](#)

definition of [A-6](#)

ALL option, SAVE command [S-2](#)

ALLINDEXESHERE, in BASETABS table [B-1](#)

ALLOCATE clause, ALTER INDEX statement [A-18](#)

ALLOCATE file attribute [A-7](#)

Allocated extents [F-19](#)

Allocating disk space with EXTENT [E-30](#)

ALLOWERRORS option

COPY command [C-116](#)

DUP command [D-72](#)

LOAD command [L-20](#)

PURGE command [P-34](#)

PURGEDATA command [P-37](#)

SECURE command [S-8](#)

ALL, in quantified predicate [Q-6](#)

ALTER CATALOG statement

description of [A-8](#)

NOPURGEUNTIL attribute [A-8](#)

OWNER file attribute [A-8](#)

SECURE file attribute [A-8](#)

ALTER COLLATION statement

description of [A-10](#)

OWNER file attribute [A-10](#)

RENAME clause [A-10](#)

SECURE file attribute [A-10](#)

ALTER DEFINE command [A-11](#)

ALTER INDEX statement

- ADD PARTITION clause [A-18](#)
- ALLOCATE clause [A-18](#)
- description of [A-13](#)
- DROP PARTITION clause [A-15](#)
- examples of [A-26](#)
- file security attributes [A-15](#)
- FIRST KEY clause [A-18](#)
- Format 2 enabling [A-25](#)
- MAXEXTENTS clause [A-18](#)
- MOVE clause [A-16](#)
- MOVE options [A-16](#)
- RENAME clause [A-15](#)
- RESETBROKEN clause [A-18](#)
- SECURE file attribute [A-15](#)
- WITH SHARED ACCESS clause [A-17](#)

ALTER PROGRAM statement

- description of [A-27](#)
- OWNER file attribute [A-27](#)
- RENAME clause [A-27](#)
- SECURE file attribute [A-27](#)

ALTER statements

- and GRANT [S-68](#)
- catalog [A-8](#)
- collation [A-10](#)
- concurrent DML operations [C-65](#)
- index [A-13](#)
- program [A-27](#)
- table [A-29](#)
- view [A-50](#)

ALTER TABLE ADD CONSTRAINT statement [S-70](#)**ALTER TABLE DROP CONSTRAINT statement** [S-70](#)**ALTER TABLE statement**

- ADD COLUMN clause [A-33](#)
- ADD PARTITION clause [A-36](#)
- DEFAULT clause [A-33](#), [D-26](#)
- description of [A-29](#)
- DROP PARTITION clause [A-34](#)

ALTER TABLE statement (continued)

- file attributes [A-32](#)
- FIRST KEY clause [A-37](#)
- Format 2 enabling [A-45](#)
- HEADING clause [H-1](#)
- MOVE clause [A-34](#)
- OWNER file attribute [A-32](#)
- PARTITION ARRAY attribute [A-33](#)
- RENAME clause [A-32](#)
- REUSE PARTITION clause [A-37](#)
- SECURE file attribute [A-32](#)
- SIMILARITY CHECK clause [A-32](#)
- splitting [A-40](#)

ALTER VIEW statement

- description of [A-50](#)
- HEADING clause [A-50](#), [H-1](#)
- OWNER file attribute [A-50](#)
- RENAME clause [A-50](#)
- SECURE file attribute [A-50](#)

Altered plan [P-22](#)**Alternate access path** [E-16](#)**Alternate message file** [Z-18](#)**AND operator** [S-5](#)**ANSI SQL, compatibility with** [S-67](#)**ANY clause, in quantified predicate** [Q-6](#)**ANYWHERE clause, INSERT statement** [I-16](#)**APPEND clause, INSERT statement** [I-16](#)**APPEND command**

- compared to COPY [A-55](#)
- compared to LOAD [A-55](#)
- description of [A-52](#)

APPENDCANCEL command [A-56](#)**APPENDRESTART command** [A-58](#)**Approximate numeric data types** [D-5](#), [N-12](#)**Arithmetic operators** [E-25](#)**AS clause**

- CREATE VIEW statement [C-167](#)
- description of [A-60](#)
- INVOKE directive [I-25](#)

AS clause (continued)
 report writer option [R-10](#)

AS DATE/TIME clause
 description of [A-67](#)
 report writer option [R-10](#)

ASCII character set [A-70](#), [C-18](#)

Associativity and UNION ALL [S-27](#)

Asterisk, in subtotal label [S-88](#)

Atomicity, statement [S-68](#)

Attributes
 See File Attributes

AUDIT
 in FILES table [F-27](#)
 in VIEWS table [V-10](#)

AUDIT file attribute
 and BUFFERED attribute [A-75](#)
 description of [A-74](#)
 set by DUP [D-78](#)
 turning off for loading data [L-18](#), [L-34](#)
 views and C [A-74](#), [C-170](#)

Audit fix-up phase [W-6](#)

Audit trails [A-75](#), [W-5](#), [W-6](#)

AUDITCOMPRESS file attribute
 ALTER INDEX statement [A-15](#)
 ALTER TABLE statement [A-32](#)
 CREATE INDEX statement [C-146](#)
 CREATE TABLE statement [C-160](#)
 description of [A-75](#)
 turned off for WITH SHARED
 ACCESS [W-6](#)

AUDITCOMPRESS, in FILES table [F-28](#)

Audited files [C-56](#)

Audited objects
 BUFFERED file attribute [B-11](#)
 description of [A-76](#)
 FREE RESOURCES statement [F-30](#)
 lock holder [L-50](#)
 lock release summary [L-48](#)
 NO SERIALWRITES file attribute [S-32](#)
 processing rules [T-7](#)

Audited objects (continued)
 rollback work [R-21](#)

Audited tables [A-76](#)

Authorization
 description of [S-11](#)
 requirements summary [S-15](#)

AUTOCOMPILE, in PROGRAMS
table [P-31](#)

AUTOWORK option
 SET SESSION command [S-39](#)
 TMF transactions [T-8](#)

AVG function [A-76](#)

B

BACKUP utility [B-1](#), [U-17](#)

Base tables
 creating [C-154](#)
 description of [T-1](#)

BASETABS catalog table [B-1](#)

BEGIN DECLARE SECTION directive [B-2](#)

Begin TMF transaction [B-3](#)

BEGIN WORK statement [B-3](#)

BETWEEN predicate [B-4](#)

BIND NAMES option, CONTROL QUERY
directive [C-74](#)

Blanks, breaking at [A-61](#)

Block buffering
 See Virtual sequential block buffering

Block size
 limit [L-6](#)
 of index [F-17](#)
 recommendation [B-5](#)

BLOCKIN option, LOAD command [L-24](#)

BLOCKOUT option, COPY
command [C-118](#)

BLOCKSIZE file attribute [B-5](#)

BLOCKSIZE, in FILES table [F-27](#)

BLOCK, parameter for FastSort [Z-4](#)

Boolean operators [S-5](#)

Break column, subtotalling [S-86](#)

BREAK FOOTING commandAS clause [A-60](#)

COMPUTE_TIMESTAMP

function [C-62](#)CONCAT clause [C-63](#)

CURRENT_TIMESTAMP

function [C-174](#)description of [B-6](#)IF/THEN/ELSE clause [I-1](#)**Break key**and CLEANUP command [C-22](#)and command files [O-2](#)and CONVERT operation [C-100](#)and COPY command [C-124](#)and DUP command [D-78](#)and LOAD command [L-33](#)and PURGE command [P-35](#)and PURGEDATA command [P-38](#)and SECURE command [S-9](#), [S-10](#)and SET SESSION command [S-43](#)effect on command files [O-2](#)effect on duplicating [D-78](#)SQLCI response to [S-40](#), [S-43](#)**BREAK KEY option, SET SESSION command** [S-40](#)**BREAK ON command**and SUBTOTAL command [S-86](#)description of [B-8](#)**BREAK TITLE command**AS clause [A-60](#)

COMPUTE_TIMESTAMP

function [C-62](#)CONCAT clause [C-63](#)

CURRENT_TIMESTAMP

function [C-174](#)description of [B-10](#)IF/THEN/ELSE clause [I-1](#)**BRIEF format, DISPLAY USE OF command** [D-54](#)**BRIEF option, FILEINFO command** [F-10](#), [F-11](#), [F-21](#)**BROWSE access**description of [A-1](#)in EXPLAIN report [E-16](#)SELECT statement [S-21](#)**BUFFERED attribute** [B-11](#), [F-27](#)**Buffered operations**INSERT operations [C-82](#)READ operations [C-82](#)UPDATE operations [C-82](#)**Buffering**

See also Virtual sequential block buffering

and INSERT [C-88](#)and READ [C-88](#)and UPDATE [C-88](#)effect on concurrency [C-88](#), [C-89](#)**C****C language**and embedded SQL [E-2](#)invoking record definition [I-26](#)**C (character) display descriptor** [A-61](#)**C89 command** [C-176](#)**Cache size** [C-134](#)**CANCEL command** [C-1](#)**CASE expression** [C-2](#)**CAST function** [C-4](#)**CATALOG clause**CREATE INDEX statement [C-144](#)CREATE TABLE statement [C-157](#)CREATE VIEW statement [C-168](#)**CATALOG command** [C-7](#)**CATALOG DEFINES** [D-32](#)**CATALOG option**CLEANUP command [C-22](#)CONVERT command [C-96](#)DUP command [D-71](#)**Catalog tables**BASSETABS [B-1](#)COLUMNS [C-46](#), [U-10](#)

Catalog tables (continued)

COMMENTS [C-51](#)
 CONSTRNT [C-70](#)
 CPRLSRCE [C-131](#)
 CPRULES [C-131](#)
 FILES [F-27](#)
 INDEXES [I-10](#)
 KEYS [K-1](#)
 operations on [C-9](#)
 PROGRAMS [P-29](#), [P-31](#)
 retrieving statistics from [U-10](#)
 See PARTNS catalog table
 See USAGES catalog table
 selecting from [S-30](#)
 TABLES [T-2](#)
 TRANSIDS [T-11](#)
 VERSIONS [V-9](#)
 VIEWS [V-10](#)

CATALOGCLASS

in CATALOGS table [C-11](#)
 in VERSIONS table [V-9](#)

CATALOGFORMAT in VERSIONS table [V-9](#)

CATALOGNAME

in CATALOGS table [C-11](#)
 in PARTNS table [P-20](#)

Catalogs

ALTER CATALOG statement [A-8](#)
 components of [C-8](#)
 CREATE CATALOG statement [C-132](#)
 creating SYSTEM CATALOG [C-152](#)
 default [Z-3](#)
 description of [C-8](#)
 displaying current default [E-3](#)
 DROP statement [D-64](#)
 expiration date, setting [N-5](#)
 format [C-8](#)
 inserting comments into [C-48](#)
 list of on node [C-11](#)
 performance considerations [C-134](#)

Catalogs (continued)

purging with CLEANUP [C-22](#)
 resecuring [A-8](#)
 setting default for [C-8](#)
 system [S-92](#)
 upgrading [U-11](#)
 versions [C-8](#), [V-7](#)

CATALOGS catalog table

creating catalog [C-132](#), [C-152](#)
 description of [C-11](#)
 securing [A-9](#)

CATALOGS Table

See CATALOGS catalog table

CATALOGVERSION

in CATALOGS table [C-11](#)
 in VERSIONS table [V-9](#)

CENTER_REPORT layout option [C-12](#), [R-10](#)

Character data

size limit [L-7](#)
 syntax [D-2](#)
 types [C-13](#)

Character expressions [C-14](#)Character literals [S-80](#)CHARACTER option, CONVERT command [C-98](#)

Character sets

ASCII listing [A-70](#)
 in collations [C-37](#)
 ISO 8859 [C-18](#)
 JIS X0208 [C-18](#)
 Kanji [C-18](#)
 KSC5601 [C-19](#)
 multibyte [M-43](#)
 shift JIS [C-18](#)
 specifying for columns [D-1](#)
 summary of [C-17](#)

Character strings

comparing [C-58](#)
 effect of collation on [C-59](#)

Character strings (continued)

literals [S-80](#)matching pattern [L-2](#)Character values, display format of [A-61](#)CHARACTERISTICS in CPRULES table [C-131](#)

CHARACTERSET

in COLUMNS table [C-47](#)in CPRULES table [C-131](#)CHAR_LENGTH function [C-20](#)

CHECK clause

CHECKMODE, in PROGRAMS table [P-32](#)CREATE CONSTRAINT statement [C-139](#)CREATE VIEW statement [C-168](#)CHECKMODE, in PROGRAMS table [P-32](#)

Clauses

AS [A-60](#)COLLATE [C-29](#)CONCAT [C-63](#)DEFAULT [D-26](#)DISTINCT [D-57](#)IF/THEN/ELSE [I-1](#)PARTITION [P-16](#)

CLEANUP command

and dependent programs [C-23](#)and shadow labels [C-22](#)description of [C-21](#)CLEAR clause, COMMENT statement [C-48](#)

CLEAR option

LOG command [L-52](#)OUT command [O-7](#)

CLEARONPURGE

in FILES table [F-27](#)in PROGRAMS table [P-31](#)

CLEARONPURGE file attribute

ALTER CATALOG statement [A-8](#)ALTER INDEX statement [A-15](#)ALTER PROGRAM statement [A-27](#)

CLEARONPURGE file attribute (continued)

description of [C-26](#)SECURE command [S-8](#)

CLOSE statement

and locking [L-51](#)description of [C-26](#)

Closing

cursors [C-27](#)FREE RESOURCES statement [F-30](#)CLUSTERING KEY clause, CREATE TABLE statement [C-158](#)

Clustering keys

and EXECUTE RETURNING [E-9](#)description of [C-28](#)length limit [L-6](#)CNVSRC file [C-97](#)

COBOL language

and embedded SQL [E-2](#)invoking record definition [I-26](#)COLCLASS, in COLUMNS table [C-46](#)

COLCOUNT

in INDEXES table [I-10](#)in TABLES table [T-2](#)

COLLATE clause

associating with a column [D-3](#)CREATE INDEX statement [C-144](#)description of [C-29](#)effect on data-type specification [D-3](#)

Collations

ALTER COLLATION statement [A-10](#)associating with a column [D-3](#)catalog description of [T-2](#)CREATE COLLATION statement [C-137](#)definitions [C-30](#)description of [C-43](#)DROP statement [D-64](#)duplicating [D-75](#)in expressions [C-14](#)renaming [A-10](#)

Collations (continued)

similarity rules for [S-57](#)

versions [V-7](#)

Collations buffer

DESCRIBE statement [D-45](#)

INCLUDE SQLDA directive [I-5](#)

Collectors [R-3](#)

COLNAME in COLUMNS table [C-46](#)

COLNUMBER in COLUMNS table [C-46](#)

COLSIZE in COLUMNS table [C-46](#)

Column identifier [C-44](#)

Columns

ADD COLUMN option [A-33](#)

assigning alias to [N-1](#)

catalog description of [C-46](#)

changing heading [A-33](#)

default values for [D-26](#)

definition of tables [C-46](#)

definition syntax [C-155](#)

description of [C-45](#)

HEADING clause [A-50](#)

heading length limit [L-6](#)

headings [D-48](#)

identifier [C-44](#)

initializing [A-33](#), [D-7](#)

limits [L-6](#)

maximum length [D-2](#)

names [C-46](#)

subtotal in report [S-86](#)

totals in report [T-9](#)

COLUMNS catalog table

description of [C-46](#)

selecting from [U-10](#)

Command files [O-1](#)

Command interface

See SQLCI

COMMAND option

LOG command [L-52](#)

SAVE command [S-3](#)

Commands

ADD DEFINE [A-4](#)

ALTER DEFINE [A-11](#)

APPEND [A-52](#)

APPENDCANCEL [A-56](#)

APPENDRESTART [A-58](#)

BREAK FOOTING [B-6](#)

BREAK ON [B-8](#)

BREAK TITLE [B-10](#)

CANCEL [C-1](#)

CATALOG [C-7](#)

CLEANUP [C-21](#)

CONVERT [C-94](#)

COPY [C-113](#)

CREATE SYSTEM CATALOG [C-152](#)

DELETE DEFINE [D-38](#)

DETAIL [D-47](#)

DISPLAY STATISTICS [D-52](#)

DISPLAY USE OF [D-54](#)

DOWNGRADE CATALOG [D-58](#)

DOWNGRADE SYSTEM

CATALOG [D-61](#)

DROP SYSTEM CATALOG [D-66](#)

DUP [D-69](#)

EDIT [E-1](#)

editing and re-executing [F-1](#)

ENV [E-3](#)

ERROR [E-4](#)

executing multiple [O-1](#)

EXIT [E-13](#)

FILEINFO [F-9](#)

FILENAMES [F-25](#)

FILES [F-26](#)

FUP [F-33](#)

GOAWAY [G-6](#)

HISTORY [H-5](#)

INFO DEFINE [I-12](#)

INITIALIZE SQL [I-13](#)

INVOKE [I-25](#)

Commands (continued)

LIST [L-16](#)
 LOAD [L-18](#)
 LOG [L-52](#)
 MODIFY CATALOG [M-4](#)
 MODIFY LABEL [M-19](#)
 MODIFY REGISTER [M-40](#)
 NAME [N-1](#)
 OUT [O-7](#)
 OUT_REPORT [O-8](#)
 PAGE FOOTING [P-1](#)
 PAGE TITLE [P-4](#)
 PERUSE [P-21](#)
 PURGE [P-33](#)
 PURGEDATA [P-36](#)
 reexecuting [Z-1](#)
 REPORT FOOTING [R-2](#)
 REPORT TITLE [R-6](#)
 RESET DEFINE [R-12](#)
 RESET LAYOUT [R-13](#)
 RESET PARAM [R-14](#)
 RESET PREPARED [R-16](#)
 RESET REPORT [R-16](#)
 RESET SESSION [R-19](#)
 RESET STYLE [R-19](#)
 SAVE [S-2](#)
 saving in a file [S-2](#)
 SECURE [S-7](#)
 SET DEFINE [S-32](#)
 SET DEFMODE [S-34](#)
 SET LAYOUT [S-34](#)
 SET PARAM [S-35](#)
 SET SESSION [S-39](#)
 SET STYLE [S-45](#)
 SHOW CONTROL [S-48](#)
 SHOW DEFINE [S-48](#)
 SHOW DEFMODE [S-49](#)
 SHOW LAYOUT [S-50](#)
 SHOW PARAM [S-50](#)

Commands (continued)

SHOW PREPARED [S-51](#)
 SHOW REPORT [S-52](#)
 SHOW SESSION [S-53](#)
 SHOW STYLE [S-54](#)
 SQLCI [S-63](#)
 SQLCOMP [S-67](#)
 SUBTOTAL [S-86](#)
 summary [S-64](#)
 SYSTEM [S-93](#)
 TEDIT [T-3](#)
 TOTAL [T-9](#)
 UPGRADE CATALOG [U-11](#)
 UPGRADE SYSTEM CATALOG [U-13](#)
 VERIFY [V-2](#)
 VOLUME [V-11](#)
 ! [Z-1](#)
 COMMENT statement [C-48](#)
 Comments
 deleting [C-49](#)
 description of [C-50](#)
 in catalogs [C-48](#)
 in collation definitions [C-30](#), [C-31](#)
 in programs and SQLCI [C-51](#)
 line length limit [L-7](#)
 number allowed per object [C-49](#)
 COMMENTS catalog table [C-51](#)
 COMMENTS option, CONVERT command [C-97](#)
 COMMENTTEXT, in COMMENTS table [C-51](#)
 COMMIT option [C-52](#)
 Commit phase [W-7](#)
 COMMIT WORK statement [C-56](#)
 Communication, SQL and host [H-6](#)
 COMPACT option, LOAD command [L-24](#)
 Comparing date-time values [C-59](#)
 Comparing numeric data [C-59](#)
 Comparing values to subquery results [Q-6](#)
 Comparison predicate, description of [C-58](#)

Compatible data types and INSERT [I-16](#)

Compilation

- authority for [S-15](#)
- automatic [P-29](#)
- explicit [P-29](#)
- Guardian command for [S-67](#)
- OSS command for [C-176](#)
- time [D-52](#)

Compiler event messages [Z-11](#)

Compilers

- and statistics [U-9](#)
- host language, versions [V-8](#)
- NonStop SQL/MP, versions [V-6](#)

Components, versions of [V-6](#)

COMPUTE_TIMESTAMP function

- and report writer [R-11](#)
- description of [C-62](#)
- SET PARAM command [S-36](#)

CONCAT clause

- description of [C-63](#)
- report writer option [R-10](#)

Concurrency

- affect of access options [A-3](#)
- and buffering [C-88](#)
- DDL and DML statements [C-65](#)
- DDL statements [D-21](#)
- description of [C-65](#)
- effect of access options [A-1](#)
- effect of VSBB [C-68](#), [C-89](#)
- utility operations [C-68](#)

Conditional

- display format [A-64](#)
- page break [D-48](#)
- printing of items [I-1](#)

Conditions for grouped rows [S-21](#)

Conditions for selecting data [S-5](#)

Configuration file

- CREATE INDEX statement [P-5](#)
- CREATEINDEX keyword [P-6](#)

Consistency, effect of access options [A-1](#)

Constraint

- catalog description of [C-70](#)
- CREATE CONSTRAINT statement [C-139](#)
- description of [C-69](#)
- DROP statement [D-64](#)
- text limit [L-7](#)
- versions [V-7](#)

CONSTRAINTNAME, in CONSTRNT table [C-70](#)

CONSTRAINTS, in BASETABS table [B-1](#)

CONSTRAINTTEXT, in CONSTRNT table [C-70](#)

CONSTRNT catalog table [C-70](#)

Continuation prompt [S-60](#)

CONTINUE statement [C-70](#)

CONTROL EXECUTOR directive [C-73](#)

Control options, displaying [S-48](#)

CONTROL QUERY directive

- BIND NAMES option [C-74](#)
- description of [C-74](#)
- HASH JOIN option [C-75](#)
- INTERACTIVE ACCESS option [C-75](#)
- MDAM option [C-75](#)

control statements [C-76](#)

CONTROL TABLE directive

- ACCESS PATH option [C-79](#)
- and BIND NAMES option [C-74](#)
- description of [C-77](#)
- displaying current values [S-48](#)
- in EXPLAIN report [E-16](#)
- JOIN METHOD option [C-80](#)
- JOIN SEQUENCE option [C-81](#)
- lock waits [C-82](#)
- MDAM option [C-81](#)
- OPEN option [C-82](#)
- RETURN IF LOCKED option [C-82](#)
- SEQUENTIAL BLOCKSPLIT option [C-83](#)
- SEQUENTIAL option [C-82](#)
- SKIP option [C-83](#)

CONTROL TABLE directive (continued)STOP AT option [C-83](#)SYNCDEPTH option [C-84](#)TABLELOCK option [C-84](#)TIMEOUT option [C-84](#)UNAVAILABLE PARTITION
option [C-83](#)with LOCK TABLE statement [L-46](#)**Controlling the SORTPROG process** [S-25](#)**Conversational interface**

See SQLCI

CONVERT commandalternate key specification [C-100](#)authorization requirement [C-99](#)behavior [C-99](#)CATALOG option [C-96](#)CHARACTER option [C-98](#)COMMENTS option [C-97](#)conversion of DDL items with [C-103](#)DDL groups [C-97](#), [C-107](#), [C-108](#)description of [C-94](#)DICTIONARY option [C-97](#)Enscribe files [C-100](#)file attributes of tables and
indexes [C-107](#)FILE IS option [C-97](#)LOAD option [C-97](#)MAP NAME option [C-96](#)NATIONAL option [C-99](#)PART option [C-97](#)partition attributes of objects [C-107](#)primary key specification [C-100](#)REDEFINE option [C-98](#)SOURCE option [C-97](#)VARCHARS option [C-97](#)Converted Enscribe applications, RETURN
IF LOCKED option [C-82](#)**Converting**binary data types [C-104](#)DDL character strings [C-104](#)DDL elementary items [C-103](#)DDL groups [C-106](#)decimal data types [C-104](#)Enscribe applications [C-82](#)Enscribe file to SQL table [C-94](#), [L-36](#)FORTRAN data types [C-105](#)SQL table to Enscribe file [L-36](#)variable-length strings [C-106](#)**CONVERTTIMESTAMP function** [C-113](#)**Copies of report** [O-9](#)**COPY command**ALLOWERRORS option [C-116](#)BLOCKOUT option [C-118](#)compared to APPEND [A-55](#)compared to LOAD [C-123](#)COUNT option [C-116](#)description of [C-113](#)display format [C-122](#)EBCDICOUT option [C-118](#)FIRST option [C-116](#)FOLD option [C-119](#)operations [C-122](#)PAD option [C-119](#)RECOUNT option [C-119](#)REPLACE SPACES WITH
option [C-116](#)REWINDOUT option [C-120](#)SHAREOUT option [C-122](#)SKIPOUT [C-120](#)SQLNULLABLE option [C-117](#)TMF transaction [C-124](#)UNLOADOUT option [C-121](#)UNSTRUCTURED option [C-117](#)UPSHIFT option [C-117](#)USESQLNULLS option [C-117](#)VAROUT option [C-121](#)

Copying

data from Enscribe file to SQL table [C-113](#)

data from SQL table to Enscribe file [C-113](#)

Enscribe files

Correlated subqueries [S-83](#)

Correlation names [C-128](#)

Cost

of executing a statement [D-52](#)

of hash operation [E-17](#)

of sort operation [E-19](#)

of SQL operation, in EXPLAIN report [E-18](#)

of total SQL statement [E-20](#)

COUNT function [C-130](#)

COUNT option

COPY command [C-116](#)

LOAD command [L-21](#)

CPARRAYENTRY in COLUMNS table [C-47](#)

CPRLSRCE catalogs table [C-131](#)

CPROBJSIZE in CPRULES table [C-131](#)

CPRULES catalog table [C-131](#)

CPRULESNAME

in COLUMNS table [C-47](#)

in CPRLSRCE table [C-131](#)

in CPRULES table [C-131](#)

in KEYS table [K-1](#)

CPUS, parameter for FastSort [Z-4](#)

CREATE ASSERTION statement [S-71](#)

CREATE CATALOG statement

description of [C-132](#)

SECURE file attribute [C-132](#)

CREATE COLLATION statement [C-137](#)

CREATE CONSTRAINT statement

CHECK clause [C-139](#)

DEFERRED [C-139](#)

description of [C-139](#)

CREATE INDEX statement

AUDITCOMPRESS file attribute [C-146](#)

CATALOG clause [C-144](#)

COLLATE clause [C-144](#)

configuration file [P-5](#)

description of [C-142](#)

file attributes [C-146](#)

INVALIDATE clause [C-144](#)

KEYTAG option [C-145](#)

PARALLEL EXECUTION clause [C-145](#)

parallel index loading [P-5](#)

PARTITION clause [C-145](#)

PHYSVOL clause [C-144](#)

UNIQUE clause [C-143](#)

WITH SHARED ACCESS clause [C-146](#)

CREATE statements and concurrent DML operations [C-65](#)

CREATE SYSTEM CATALOG command [C-152](#)

CREATE TABLE statement

AUDITCOMPRESS file attribute [C-160](#)

CATALOG option [C-157](#)

CLUSTERING KEY clause [C-158](#)

DEFAULT clause [C-157](#), [D-26](#)

description of [C-154](#)

file attributes [C-160](#)

FORMAT file attribute [C-161](#)

HEADING clause [C-157](#), [H-1](#)

LIKE clause [C-155](#)

NOT NULL clause [C-157](#)

ORGANIZATION clause [C-158](#)

PARTITION ARRAY clause [C-159](#)

PARTITION clause [C-158](#)

PHYSVOL clause [C-158](#)

PRIMARY KEY clause [C-157](#)

SECURE file attribute [C-159](#)

SIMILARITY CHECK clause [C-160](#)

CREATE VIEW statement

- AS clause [C-167](#)
- CATALOG clause [C-168](#)
- CHECK [C-168](#)
- description of [C-166](#)
- FOR PROTECTION clause [C-168](#)
- HEADING clause [C-167](#), [H-1](#)
- length of [C-170](#)
- SECURE file attribute [C-168](#)
- SELECT DISTINCT clause [S-68](#)
- SIMILARITY CHECK clause [C-168](#)
- WITH CHECK OPTION clause [C-169](#)
- WITH HEADINGS clause [C-169](#)
- WITH HELP TEXT clause [C-169](#)

CREATEINDEX keyword, configuration file [P-6](#)**CREATETIME**

- in INDEXES table [I-10](#)
- in PROGRAMS table [P-31](#)
- in TABLES table [T-2](#)

Creating

- catalog [C-132](#)
- CATALOGS table [C-152](#)
- collation [C-137](#)
- constraint [C-139](#)
- table [C-154](#)
- view [C-166](#)

CTOEDIT command [F-8](#)**CURRENT default value** [D-26](#)**CURRENT function** [C-173](#)**CURRENT_TIMESTAMP function**

- and report writer [R-11](#)
- description of [C-174](#)
- in USING clause, EXECUTE statement [E-8](#)
- SET param command [S-35](#)

Cursors

- CLOSE statement [C-26](#)
- closing [C-27](#)
- declaration [D-23](#)

Cursors (continued)

- description of [C-175](#)
- effect of FREE RESOURCES statement [F-30](#)
- execution time [C-74](#)
- FETCH statement [F-3](#), [F-4](#)
- limits [L-7](#)
- OPEN statement [O-5](#)
- operations with VSBB [C-91](#)
- position [C-176](#), [D-39](#)
- stability [C-176](#)
- updating by position [U-4](#)

D**Damaged objects, deleting** [C-21](#)**Data**

- clearing from file or table [D-39](#), [P-36](#)
- compression of index blocks [I-1](#)
- copying [C-113](#)
- fetching [F-3](#)
- inserting [I-14](#)
- loading into database [L-18](#)
- modifying [U-3](#)
- removing [C-26](#)
- selecting [S-17](#)
- size limits [L-7](#)
- updating [U-3](#)

Data Control Language (DCL)

- authorization [S-15](#)
- description of [D-18](#)

Data Control Language (DCL) statements

- CONTROL EXECUTOR [C-73](#)
- CONTROL QUERY [C-74](#)
- CONTROL TABLE [C-77](#)
- description of [D-18](#)
- FREE RESOURCES [F-30](#)
- LOCK TABLE [L-45](#)
- UNLOCK TABLE [U-1](#)

Data declaration

BEGIN DECLARE SECTION

directive [B-2](#)

END DECLARE SECTION

directive [E-3](#)tables and views [I-25](#)

Data Definition Language (DDL)

alternate keys [C-100](#)and database concurrency [C-65](#), [D-21](#)clauses, CONVERT statement [C-101](#)groups, converting [C-107](#)performance considerations [C-134](#)primary key [C-100](#)record definitions, Enscribe file [C-100](#)security requirements [S-15](#)

Data Definition Language (DDL) statements

ALTER CATALOG [A-8](#)ALTER COLLATION [A-10](#)ALTER INDEX [A-13](#)ALTER PROGRAM [A-27](#)ALTER TABLE [A-29](#)ALTER VIEW [A-50](#)COMMENT [C-48](#)CREATE CATALOG [C-132](#)CREATE COLLATION [C-137](#)CREATE CONSTRAINT [C-139](#)CREATE TABLE [C-154](#)CREATE VIEW [C-166](#)description of [D-20](#)DROP [D-63](#)HELP TEXT [H-4](#)processing rules [T-7](#)UPDATE STATISTICS [U-7](#)Data dictionary [D-1](#)

Data Management Language (DML)

statements [C-65](#)

Data Manipulation Language (DML)

statements [D-58](#)Data Manipulation Language (DML)
statementsand database concurrency [C-65](#)authorization requirements [S-16](#)processing rules [T-7](#)

Data retrieval

cursor declaration [D-23](#)SELECT statement [S-17](#)

Data selection

See Selecting data

Data Status Language (DSL)

statements [D-67](#)

Data Status Language (DSL) statements

GET CATALOG OF SYSTEM [G-1](#)GET VERSION [G-2](#)GET VERSION OF PROGRAM [G-4](#)

Data syntax

CHARACTER [D-2](#)DATETIME [D-6](#)DECIMAL [D-5](#)DOUBLE PRECISION [D-5](#)FLOAT [D-5](#)INTEGER [D-5](#)INTERVAL [D-7](#)LARGEINT [D-5](#)NUMERIC [D-4](#)PICTURE 9 [D-6](#)PICTURE X [D-4](#)REAL [D-5](#)SMALLINT [D-5](#)

Data types

character [C-13](#)converting [C-4](#), [L-36](#)DATE [D-8](#)DATETIME [D-15](#)date-time [D-9](#)DDL [L-36](#)decimal [C-104](#)description of [D-1](#)dynamic SQL parameters and [P-13](#)

Data types (continued)

for parameters [C-4](#), [P-14](#), [S-36](#)FORTRAN [C-105](#)INTERVAL [I-19](#)Julian timestamp [C-62](#)numeric [N-11](#)TIME [T-5](#)TIMESTAMP [T-5](#)view columns [C-170](#)

Database

authorization for access [S-11](#)integrity [C-69](#), [C-139](#)sample [S-1](#)updating statistics [U-7](#)DATATYPE in COLUMNS table [C-46](#)

Date

computing for report [C-62](#), [C-173](#)of expiration [A-32](#)print item display format [A-67](#)specifying default format [D-8](#)DATE data type [D-8](#)DATE literal [D-11](#)

DATE values

in host variables [H-6](#)LOAD command [L-38](#)DATEFORMAT function [D-14](#)

DATETIME

data syntax [D-6](#)data type [D-15](#)literals [D-11](#)values, LOAD command [L-38](#)DATETIMEENDFIELD in COLUMNS table [C-47](#)DATETIMEEQUALIFIER in COLUMNS table [C-47](#)DATETIMESTARTFIELD in COLUMNS table [C-47](#)

Date-time

arithmetic [E-25](#)comparing [C-59](#)

Date-time (continued)

functions [D-10](#)in host variables [H-6](#)inserting [I-17](#), [I-18](#)items, in expressions [E-25](#)literals [D-10](#)

Date-time data types

description of [D-9](#)TIME [D-11](#)TIMESTAMP [D-11](#)

Date-time expressions

description of [E-22](#)evaluation [E-25](#)DATE_FORMAT layout option [D-8](#), [R-10](#)DAYOFWEEK function [D-17](#)

DCL

See Data Control Language(DCL)

DCOM utility [U-17](#)

DCOMPRESS file attribute

description of [D-18](#)displaying [F-17](#)DCOMPRESSTYPE, in FILES table [F-28](#)DCOMPRESS, in FILES table [F-27](#)

DDL

See Data Definition Language (DDL)

DEADLOCKS [D-22](#)DEALLOCATE file attribute [A-7](#)DECIMAL columns, explicit lengths [S-69](#)DECIMAL data syntax [D-5](#)DECIMAL data types, converting to SQL [C-104](#)DECIMAL_POINT layout option [D-22](#), [R-10](#)

DECLARE CURSOR statement

and locking [D-25](#)description of [D-23](#)FOR UPDATE OF clause [D-23](#)

Declare section

declaration [B-2](#)terminating [E-3](#)

Decorations

- default [A-64](#)
- description of [A-64](#)
- display [A-60](#)
- using [A-64](#)

Dedicated-operation-in-progress prompt [S-60](#)

Default

- catalog [C-8](#)
- column values [D-26](#)
- CURRENT value [D-26](#)
- decimal character [D-22](#)
- decorations [A-64](#)
- detail line [D-50](#)
- displaying current [E-3](#)
- layout options [R-13](#)
- line length [L-7](#)
- names in prepared commands [P-25](#)
- node, displaying [E-3](#)
- page length for reports [P-2](#)
- print item headings [D-50](#)
- session options [R-19](#)
- space between print items [S-58](#)
- style options [R-19](#)
- subvolume [V-11](#)
- system [D-27](#), [E-3](#)
- time display format [T-4](#)
- value, for column [D-7](#)
- volume [V-11](#)

DEFAULT clause

- ALTER TABLE statement [A-33](#)
- CREATE TABLE statement [C-157](#)
- description of [D-26](#)

DEFAULTCLASS in COLUMNS table [C-47](#)DEFAULTS DEFINES and catalogs [C-8](#), [D-32](#)DEFAULTVALUE in COLUMNS table [C-47](#)

DEFINES

- and SQLCI [D-28](#)
- and SQLCI HELP [D-28](#)

DEFINES (continued)

- as logical names [D-27](#)
- attributes [D-32](#)
- CLASS [D-32](#)
- DEFMODE attribute [D-28](#)
- description of [D-27](#)
- Guardian procedure calls [D-31](#)
- propagation of [D-28](#)
- rules for naming [A-4](#)
- saving in file [S-2](#)
- summary of commands [D-29](#)
- summary of procedures [D-29](#)
- system [S-94](#)
- using from SQLCI [D-30](#)
- using with SQL programs [D-28](#)
- working attribute set [D-32](#), [D-34](#)

Defines

- =_AUDSERV_XSWAP_node [Z-2](#)
- =_DEFAULTS [Z-3](#)
- =_SORT_DEFAULTS [Z-4](#)
- =_SQL_AUD_node [Z-14](#)
- =_SQL_CAT_HEAP_LIMIT [Z-6](#)
- =_SQL_CAT_node [Z-14](#)
- =_SQL_CI2_node [Z-14](#)
- =_SQL_CMP_CPUS_node [Z-7](#)
- =_SQL_CMP_DOUBLE_SBB_OFF [Z-8](#)
- =_SQL_CMP_DOUBLE_SBB_ON [Z-9](#)
- =_SQL_CMP_EQ_LIMIT [Z-10](#)
- =_SQL_CMP_EVENT [Z-11](#)
- =_SQL_CMP_EVENT_NO0 [Z-12](#)
- =_SQL_CMP_node [Z-14](#)
- =_SQL_cmp_node [Z-14](#)
- =_SQL_CMP_NO_KS_MJOIN [Z-14](#)
- =_SQL_EXE_DOUBLE_SHUTOFF [Z-15](#)
- =_SQL_EXE_ESPS_CK_CM0N [Z-16](#)
- =_SQL_EXE_USE_SWAPVOL [Z-17](#)
- =_SQL_MSG_node [Z-18](#)
- =_SQL_RECGEN_node [Z-20](#)

Defines (continued)

=_SQL_TM_node_vol [Z-20](#)

=_SQL_UTL_node [Z-14](#)

DEFINES option, SAVE command [S-2](#)

DEFMODE attribute [D-28](#)

DELETE DEFINE command [D-38](#)

DELETE statement

description of [D-39](#)

lock release summary [L-48](#)

table privileges [S-68](#)

WHERE clause [D-39](#)

WHERE CURRENT OF clause [D-39](#)

Deleting

damaged objects [C-21](#)

help text [H-4](#)

rows [D-39](#)

system catalog [D-66](#)

Delimited identifiers [S-69](#)

Dependent object

DISPLAY USE OF command [D-54](#)

information in USAGES table [U-15](#)

type [D-55](#)

DESCRIBE INPUT statement [D-41](#)

DESCRIBE statement

collations buffer [D-45](#)

description of [D-43](#)

names buffer [D-44](#)

Detail alias

description of [D-46](#)

in DETAIL command [D-49](#)

not allowed in DETAIL command [D-47](#)

DETAIL command

AS clause [A-60](#)

COMPUTE_TIMESTAMP
function [C-62](#)

CONCAT clause [C-63](#)

CURRENT_TIMESTAMP
function [C-174](#)

description of [D-47](#)

IF/THEN/ELSE clause [I-1](#)

DETAIL command (continued)

NAME option [D-49](#)

NEED clause [D-49](#)

PAGE clause [D-49](#)

SKIP clause [D-49](#)

SPACE clause [D-49](#)

TAB clause [D-49](#)

DETAIL display, FILEINFO command [F-15](#)

Detail line

controlling division of default [L-53](#)

default [D-50](#)

grouping with others [B-8](#)

printing sequence number of [L-14](#)

specifying [D-47](#)

vertical printing of [D-51](#)

DETAIL option

ERROR command [E-5](#)

FILEINFO command [F-10](#), [F-11](#), [F-14](#),
[F-21](#)

Device, output [R-20](#), [S-42](#)

DICTIONARY option, CONVERT
command [C-97](#)

Dictionary, SQL/MP [D-1](#)

Directing output to a file [O-7](#)

Directives

BEGIN DECLARE [B-2](#)

CONTROL EXECUTOR [C-73](#)

CONTROL QUERY [C-74](#)

CONTROL TABLE [C-77](#)

description of [S-73](#)

END DECLARE SECTION [E-3](#)

EXPLAIN [E-13](#)

INCLUDE SQLCA [I-4](#)

INCLUDE SQLDA [I-4](#)

INCLUDE SQLSA [I-6](#)

INCLUDE STRUCTURES [I-7](#)

INVOKE [I-25](#)

SQL [S-59](#)

Disk

erasing [C-26](#)

Disk file

- device width used [R-20](#)
- labels in SQL [D-1](#)

Disk space

- conserving [D-18](#)
- manipulating [A-7](#)

Display formats

- date print item [A-67](#)
- default, for date [D-8](#)
- default, for time [T-4](#)
- print item [A-60](#)
- STATISTICS of FILEINFO [F-22](#)
- subtotal [S-86](#)
- time print item [A-67](#)
- total [T-9](#)

Display format, COPY command [C-122](#)Display modifiers [A-60](#)DISPLAY STATISTICS command [D-52](#)

DISPLAY USE OF command

- AT option [D-54](#)
- authorization requirements [D-54](#)
- brief format [D-54](#)
- description of [D-54](#)
- USAGES table [D-55](#)

Displaying

- contents of a file [C-113](#)
- control options [S-48](#)
- current default [E-3](#)
- current default catalog [E-3](#)
- current default volume [E-3](#)
- current TMF transaction ID [E-3](#)
- current TMF transaction status [E-3](#)
- DCOMPRESS attribute [F-17](#)
- default node [E-3](#)
- Enscribe files [C-123](#)
- environmental attributes [E-3](#)
- error [S-40](#)
- file attributes [F-18](#)
- ICOMPRESS attribute [F-17](#)

Displaying (continued)

- keys [F-17](#)
- length of keys column [F-17](#)
- levels of index [F-19](#)
- LOCKLENGTH attribute [F-17](#)
- name of LOG file [E-3](#)
- name of MESSAGEFILE [E-3](#)
- name of OUT file [E-3](#)
- object [D-54](#), [F-15](#)
- OUT_REPORT [E-3](#)
- ownership [F-12](#)
- ownership of object [D-56](#)
- PARTITION ARRAY attribute [F-20](#)
- physical characteristics [F-9](#)
- record length [F-16](#)
- security [F-12](#)
- security of object [D-56](#)
- software version [E-3](#)
- statistics [S-41](#)
- statistics for command [D-52](#)
- warning messages [S-42](#)
- DISPLAY_ERROR option, SET SESSION command [S-40](#)
- DISTINCT clause
 - and null values [N-9](#)
 - description of [D-57](#)
 - SELECT statement [S-19](#)
- Distinct rows, selecting [S-19](#)
- Distributed Systems Management (DSM) [R-3](#)
- DML
 - See Data Manipulation Language (DML)
- DOUBLE PRECISION data syntax [D-5](#)
- Double spacing [L-16](#)
- DOWNGRADE CATALOG command [D-58](#)
- DOWNGRADE SYSTEM CATALOG command [D-61](#)
- DROP ASSERTION statement [S-71](#)
- Drop objects [D-63](#)

DROP PARTITION clause, ALTER INDEX statement [A-15](#)

DROP PARTITION clause, ALTER TABLE statement [A-34](#)

DROP statement [D-63](#)

DROP SYSTEM CATALOG command [D-66](#)

DSAP utility [U-17](#)

DSL

See Data Status Language (DSL)

DSLACK file attribute [D-68](#)

DSLACK option, LOAD command [L-28](#)

DSM (Distributed Systems Management) [R-3](#)

DUP command

authorization requirements [D-74](#)

CATALOG option [D-71](#)

description of [D-69](#)

INDEXES option [D-74](#)

LISTALL option [D-72](#)

MAP NAME option [D-70](#)

rules [D-74](#)

SAVEALL option [D-73](#)

SAVEID option [D-73](#)

SOURCEDATE option [D-73](#)

TARGET option [D-73](#)

VIEW option [D-74](#)

Duplicate values and unique index [C-143](#)

Duplicating Enscribe files [D-76](#)

Duplicating with DUP command [D-69](#)

Dynamic SQL statements

cursor declaration [D-23](#), [D-24](#)

describing input parameters [D-41](#)

describing output parameters [D-43](#)

describing output variables [D-43](#)

immediate execution [E-11](#)

Dynamic SQL, description of [D-80](#)

D> prompt [S-60](#)

E

EBCDICIN option, LOAD command [L-24](#)

EBCDICOUT option, COPY command [C-118](#)

EDIT command [E-1](#)

Edit files [F-8](#)

Editing a command [F-1](#)

Editor

EDIT [E-1](#)

TEDIT [T-3](#)

vi [F-8](#)

Efficiency of multivalue predicates [C-60](#)

Elapsed compilation time [D-52](#)

ELSE clause [I-1](#)

Embedded SQL [E-2](#)

EMPTYOK option, LOAD command [L-21](#)

EMS

See Event Management Service

EMS default reports [R-3](#)

END DECLARE SECTION directive [E-3](#)

Ending TMF transaction [C-56](#)

ENDTRANSACTION procedure call [C-57](#)

Enscribe applications, converted [C-82](#)

Enscribe files

clearing data from [P-36](#)

converting to an SQL table [C-94](#)

copying to an SQL table [C-113](#), [C-125](#)

DDL record definition [C-100](#)

description of [F-8](#)

displaying [C-123](#), [F-9](#)

duplicating [D-76](#)

loading data into [L-34](#)

Entry-sequenced files [F-8](#)

Entry-sequenced tables [S-90](#)

ENV command [E-3](#)

ENV option, SAVE command [S-2](#)

Environmental attributes

displaying [E-3](#)

saving in a file [S-2](#)

EOF

address of file or object [F-19](#)

and filesets [Q-3](#)

in FILES table [F-28](#)

Erasing data from disk [C-26](#)

ERROR command

description of [E-4](#)

DETAIL option [E-5](#)

Error messages

description of [E-6](#)

FETCH statement [F-4](#)

Errors

10088 [S-44](#)

1057 [C-54](#), [D-21](#)

1125 [A-24](#), [A-45](#)

12 [C-54](#), [D-21](#)

1203 [D-21](#)

122 [C-88](#), [Z-17](#), [Z-21](#)

1222 [D-21](#)

1411 [A-25](#)

1615 - 1618 [C-54](#)

1621 - 1622 [C-54](#)

199 (Disk file is Safeguard protected) [A-28](#)

200 - 255 (A path or network error occurs) [C-83](#)

3001 - 3999 [C-54](#)

35 (Lock limit has been reached) [C-84](#)

40 [C-54](#), [D-21](#), [D-22](#), [P-35](#), [P-39](#)

40 (Operation timed out) [C-85](#)

45 [A-17](#), [A-35](#)

45 (File is full) [P-9](#)

59 (The file is bad) [C-83](#)

60 [C-147](#)

61 (No more opens are permitted on the volume) [C-83](#)

66 (The volume is not available) [C-83](#)

73 [C-54](#), [C-82](#), [D-21](#), [D-22](#), [P-35](#), [P-38](#), [T-7](#)

73 (File/Record locked) [C-85](#)

8204 [C-147](#)

9179 [A-57](#)

9182 [A-57](#)

Errors (continued)

A path or network error occurs (200 - 255) [C-83](#)

checking with SQLCA [I-4](#)

Disk file is Safeguard protected (199) [A-28](#)

displaying [S-40](#)

file system [D-21](#)

File/Record locked (73) [C-85](#)

Lock limit has been reached (35) [C-84](#)

No more opens are permitted on the volume (61) [C-83](#)

Operation timed out (40) [C-85](#)

sort error 30 [P-9](#)

SQL/MP [D-21](#)

testing for with WHENEVER [W-1](#)

The file is bad (59) [C-83](#)

The volume is not available (66) [C-83](#)

-3036 [M-43](#)

-6021 [C-80](#)

-8300 [C-82](#)

-9132 [F-11](#)

-9133 [F-11](#)

-9853 [V-4](#)

ERROR_ABORT option, SET SESSION command [S-40](#)

ERROR_TEXT option, SET SESSION command [S-41](#)

Escape characters [L-3](#)

ESPs, in EXPLAIN report [E-18](#)

Estimated execution cost [D-52](#)

Evaluation of arithmetic expressions [E-21](#)

Event Management Service (EMS)

and online dumps of target objects [W-6](#)

description of [R-3](#)

messages from SQL operations [R-3](#)

messages from SQLCOMP

compiler [Z-11](#), [Z-13](#)

Event messages, from SQLCOMP compiler [Z-11](#), [Z-13](#)

Exact numeric data types [N-11](#), [N-12](#)

Examples, database used in [S-1](#)
 Exclamation point command [Z-1](#)
 EXCLUSIVE clause, LOCK TABLE statement [L-45](#)
 Exclusive locks
 description of [L-50](#)
 EXCLUSIVE clause [L-45](#)
 on SELECT statement [S-21](#)
 Execute access [S-14](#)
 EXECUTE command
 COMPUTE_TIMESTAMP function [C-62](#)
 CURRENT_TIMESTAMP function [C-174](#)
 EXECUTE IMMEDIATE statement [E-11](#)
 EXECUTE statement
 CURRENT_TIMESTAMP function [E-8](#)
 description of [E-7](#)
 RETURNING clause [E-9](#)
 USING clause [E-7](#)
 USING DESCRIPTOR clause [E-7](#)
 Execution plans [P-22](#)
 EXISTS predicate [E-12](#)
 EXIT command [E-13](#)
 Expiration date [A-32](#), [N-5](#)
 EXPLAIN directive [E-13](#)
 Exploded record length [F-16](#)
 Exponents [N-13](#)
 Expressions
 CASE [C-2](#)
 character [C-14](#)
 date-time [E-22](#)
 date-time, evaluation [E-25](#)
 description of [E-21](#)
 INTERVAL [E-22](#)
 numeric [E-22](#)
 with date-time items [E-25](#)
 EXTEND function [E-28](#)
 EXTENT file attribute [E-30](#)
 Extents
 Available space in file or object [F-19](#)

Extents (continued)
 MAXEXTENTS file attribute [M-2](#)
 EXTENTS ALLOCATED display, FILEINFO command [F-19](#)
 EXTENTS display, FILEINFO command [F-21](#)
 EXTENTS option, FILEINFO command [F-10](#)

F

FastSort [Z-4](#)
 FC command, prompt [S-60](#)
 FETCH statement
 and cursors [F-4](#)
 and SQLDA [F-3](#)
 description of [F-3](#)
 INTO clause [F-3](#)
 lock release summary [L-48](#)
 lock release summary [L-48](#)
 USING DESCRIPTOR clause [F-3](#)
 Fetching rows [F-3](#)
 Field conversion [C-126](#), [L-36](#)
 Field formats
 and copying data [C-126](#)
 and loading data [L-36](#)
 File attributes
 ALLOCATE [A-7](#)
 ALTER INDEX statement [A-15](#)
 ALTER TABLE statement [A-32](#)
 AUDIT [A-74](#)
 AUDITCOMPRESS [A-75](#)
 BLOCKSIZE [B-5](#)
 BUFFERED [B-11](#)
 CLEARONPURGE [C-26](#)
 CREATE INDEX statement [C-146](#)
 CREATE TABLE statement [C-160](#)
 DCOMPRESS [D-18](#)
 DEALLOCATE [A-7](#)
 description of [F-6](#)
 displaying [F-18](#)

File attributes (continued)

DSLACK [D-68](#)
 EXTENT [E-30](#)
 ICOMPRESS [I-1](#)
 ISLACK [I-29](#)
 LOCKLENGTH [L-51](#)
 MAXEXTENTS [M-2](#)
 NOPURGEUNTIL [N-5](#)
 OWNER [O-10](#)
 PROGID [P-28](#)
 RECLENGTH [R-1](#)
 RESETBROKEN [R-20](#)
 SECURE [S-11](#)
 SERIALWRITES [S-32](#)
 SLACK [S-57](#)
 TABLECODE [T-1](#)
 VERIFIEDWRITES [F-30](#), [V-1](#)

FILE IS option, CONVERT command [C-97](#)

File labels, used by SQL [D-1](#)

File names

Guardian [G-7](#)
 simple [G-7](#)

File organizations [F-8](#)

File Utility Program (FUP) [F-33/F-36](#)

FILEFORMAT, in FILES table [F-28](#)

FILEINFO command

BRIEF option [F-10](#), [F-11](#), [F-21](#)
 description of [F-9](#)
 DETAIL display [F-15](#)
 DETAIL display for OSS files [F-21](#)
 DETAIL display for views [F-20](#)
 DETAIL option [F-10](#), [F-11](#)
 DETAIL option for objects [F-14](#)
 EXTENTS display [F-19](#), [F-21](#)
 EXTENTS option [F-10](#)
 STATISTICS display [F-22](#)
 STATISTICS option [F-10](#)
 VERSION option [F-16](#)

FILENAME

in BASETABS table [B-1](#)

in FILES table [F-27](#)

in INDEXES table [I-10](#)

in PARTNS table [P-20](#)

FILENAMES command [F-25](#)

Files

attributes of [F-26](#)
 available space [F-19](#)
 CNVSRC [C-97](#)
 code [F-8](#), [T-1](#)
 directing output to [O-7](#)
 displaying contents of [C-113](#)
 displaying name of [F-25](#)
 edit [F-8](#)
 Enscribe [F-8](#)
 entry-sequenced [F-8](#)
 key-sequenced [F-8](#)
 LOG [L-52](#), [Z-12](#)
 logging to [L-52](#)
 MESSAGE [M-3](#), [Z-18](#)
 OBEY [O-1](#)
 open state [F-11](#), [F-19](#)
 organization of [F-8](#)
 OUT_REPORT [O-8](#)
 ownership [S-14](#)
 position of [C-176](#)
 relative [F-8](#)
 set size [E-30](#)
 SQL messages [Z-18](#)
 statistics [F-22](#)
 structured [F-8](#)
 temporary, size limit [L-13](#)
 unstructured [F-8](#)
 FILES catalog table [F-27](#)
 FILES command [F-26](#)
 Filesets
 description of [F-29](#)
 qualified list [Q-1](#)

- FILETYPE, in FILES table [F-27](#)
- File, formats, Enscribe [Q-4](#)
- File-system errors [D-21](#)
- Filler characters, overflow [O-10](#)
- FIRST KEY clause
 - ALTER INDEX statement [A-18](#)
 - ALTER TABLE statement [A-37](#)
 - defaults for unspecified columns [P-17](#)
- First key, size limit [L-8](#)
- FIRST option
 - COPY command [C-116](#)
 - LOAD command [L-21](#)
- FIRSTKEY, in PARTNS table [P-20](#)
- Fixing command [F-1](#)
- FLOAT data syntax [D-5](#)
- FOLD option, COPY command [C-119](#)
- Folding a line [L-53](#)
- Footing
 - See PAGE FOOTING command
- Footings
 - break [B-6](#)
 - page [P-1](#)
 - report [R-2](#)
- FOR PROTECTION clause, CREATE VIEW statement [C-168](#)
- FOR UPDATE OF clause
 - DECLARE CURSOR statement [D-23](#)
 - SELECT statement [S-24](#)
- Forced page break [D-49](#)
- FORCE, in PROGRAMS table [P-32](#)
- FOREIGN KEY table constraint [S-68](#)
- Form feeds for reports [P-3](#)
- Form name for reports [O-9](#)
- Format
 - dates and times [C-174](#)
 - Enscribe files [Q-4](#)
 - file [F-30](#)
 - print item [A-60](#)
 - type [A-25](#)
- Format 2 partitions
 - description of [F-30](#)
 - EXTENT sizes [E-30](#)
 - FORMAT file attribute [F-30](#)
 - limits [L-9](#)
 - MAXEXTENTS [M-2](#)
 - migration [A-46](#)
 - PARTITION clause [P-16](#)
- Format 2-enabled
 - indexes [A-25](#), [C-149](#)
- Format 2-enabled tables
 - command [C-156](#)
 - example [C-163](#), [C-165](#)
 - partition [A-47](#)
 - partition array [A-33](#), [A-45](#), [C-159](#)
 - partition array parameter [C-162](#)
 - partition formats [C-161](#)
- FORMAT file attribute [F-30](#)
- Formatting commands
 - See also Report
- FORTTRAN data types, converting to SQL [C-105](#), [S-69](#)
- FREE RESOURCES statement
 - and COMMIT WORK statement [C-57](#)
 - and ROLLBACK WORK statement [R-22](#)
 - description of [F-30](#)
 - lock release summary [L-48](#)
 - lock release summary [L-48](#)
- FROM clause
 - limit on tables [L-8](#)
 - SELECT statement [S-20](#)
- FSDATATYPE in COLUMNS table [C-46](#)
- Functions
 - aggregate [A-6](#), [F-32](#)
 - AVG [A-76](#)
 - CAST [C-4](#)
 - CHAR_LENGTH [C-20](#)
 - COMPUTE_TIMESTAMP [C-62](#)
 - CONVERTTIMESTAMP [C-113](#)

Functions (continued)

COUNT [C-130](#)
 CURRENT [C-173](#)
 CURRENT_TIMESTAMP [C-174](#)
 DATEFORMAT [D-14](#)
 date-time [D-10](#)
 DAYOFWEEK [D-17](#)
 description of [F-32](#)
 EXTEND [E-28](#)
 JULIANTIMESTAMP [J-4](#)
 LINE NUMBER [L-14](#)
 MAX [M-1](#)
 MIN [M-3](#)
 PAGE_NUMBER [P-3](#)
 POSITION [P-23](#)
 scalar [F-33](#), [S-46](#)
 SETSCALE [S-46](#)
 string [S-80](#)
 SUBSTRING [S-84](#)
 SUM [S-89](#)
 TRIM [T-11](#)
 UPSHIFT [U-14](#)
 FUP command [F-33/F-36](#)
 FUP LICENSE command [F-35](#)

G

Generalized owner [G-1](#), [S-9](#), [S-14](#)
 Generic lock [L-51](#)
 GET CATALOG OF SYSTEM
 statement [G-1](#)
 GET VERSION OF PROGRAM
 statement [G-4](#)
 GET VERSION statement [G-2](#)
 GOAWAY command [G-6](#), [U-17](#)
 GRANT statement [S-68](#)
 Granularity [L-49](#)
 GROUP BY clause
 and null values [N-9](#)
 and UNION operator [S-27](#)
 SELECT statement [S-22](#), [S-25](#)

Group manager [G-7](#), [S-12](#)
 Group number [S-12](#)
 Grouped rows, conditions [S-21](#)
 Grouped view
 description of [S-24](#)
 FROM clause [S-19](#)
 view creation [C-170](#)
 GROUPID
 in PROGRAMS table [P-31](#)
 in TABLES table [T-2](#)
 Groups
 and GROUP BY clause [S-22](#)
 and SECURE file attribute [S-11](#)
 and SELECT statement [S-27](#)
 Guardian names [G-7](#)
 Guardian processes [E-2](#)
 Guardian user IDs [S-12](#)

H

HASH JOIN option, CONTROL QUERY
 directive [C-75](#)
 Hash joins
 restrictions on [C-80](#)
 with CONTROL QUERY directive [C-75](#)
 HAVING clause
 and UNION operator [S-27](#)
 SELECT statement [S-21](#)
 HEADING clause
 ALTER VIEW statement [A-50](#)
 CREATE TABLE statement [C-157](#)
 CREATE VIEW statement [C-167](#)
 description of [H-1](#)
 HEADING in COLUMNS table [C-47](#)
 Headings
 concatenated print items [C-64](#)
 multiple line [N-4](#)
 print item [D-48](#)
 print item default [D-50](#)
 suppressing in reports [H-1](#)
 HEADINGS layout option [H-2](#), [R-10](#)

HEADINGTEXT in COLUMNS table [C-47](#)
 HELP command [H-2](#)
 HELP TEXT statement [H-4](#)
 Help text, deleting [H-4](#)
 Hidden text [C-51](#)
 HISTORY command [H-5](#)
 Host and SQL communication [H-6](#)
 Host identifiers [H-5](#)
 Host object SQL version (HOSV) [V-8](#)
 Host programming language
 compiler versions [V-8](#)
 description of [E-2](#), [H-5](#)
 Host programs [H-6](#)
 Host variables
 declaration [B-2](#)
 description of [H-6](#)
 substituting values in a query [H-6](#)
 TYPE AS clause [H-6](#)
 HOSV (host object SQL version) [V-8](#)

I

ICOMPRESS file attribute
 description of [I-1](#)
 displaying [F-17](#)
 ICOMPRESS, in FILES table [F-27](#)
 Identifiers, SQL [S-59](#)
 IDs [S-12](#)
 IF/THEN/ELSE clause
 description of [I-1](#)
 report writer option [R-10](#)
 IN predicate [I-3](#), [L-8](#)
 INCLUDE SQLDA directive
 collation buffer [I-5](#)
 description of [I-4](#)
 names buffer [I-5](#)
 INCLUDE SQLSA directive [I-6](#)
 INCLUDE STRUCTURES directive [I-7](#)
 Index
 address of EOF [F-19](#)
 ALTER INDEX statement [A-13](#)

Index (continued)
 altering attributes [A-15](#)
 alternate access path [E-16](#)
 available space [F-19](#)
 block length [F-17](#)
 catalog description of [I-10](#)
 column limit [L-6](#)
 configuration file for parallel loading [P-5](#)
 CREATE INDEX statement [C-142](#)
 data, loading into [L-18](#)
 date-caused program recompilation [F-19](#)
 dependencies on base table [C-148](#)
 displaying physical characteristics [F-9](#)
 DROP statement [D-64](#)
 dropping [P-33](#)
 expiration date, setting [N-5](#)
 file statistics [F-22](#)
 for catalog tables [C-9](#)
 keytags [I-9](#)
 levels and ICOMPRESS [I-1](#)
 limit per table [L-8](#)
 open state [F-19](#)
 partitioned [P-19](#)
 physical file attributes [C-107](#)
 size limit [L-8](#)
 sorting and null values [C-149](#)
 specifying file attributes of an [C-146](#)
 unique and nonunique [I-9](#)
 unpartitioned [P-19](#)
 using subsorts [C-145](#)
 versions [V-7](#)
 Index keys [I-9](#)
 Index loading
 partitions in parallel [C-145](#), [P-5](#)
 using subsorts [C-151](#)
 Index partitions
 adding [A-18](#)
 null values [P-17](#)

- INDEXES catalog table [I-10](#)
- INDEXES option, DUP command [D-74](#)
- INDEXLEVELS, in INDEXES table [I-10](#)
- INDEXNAME
 - in INDEXES table [I-10](#)
 - in KEYS table [K-1](#)
- INDICATOR clause
 - host variable [H-6](#)
 - parameter name [P-12](#)
- INDICATOR parameter [I-11](#)
- Indicator variable [H-6](#), [I-11](#), [P-12](#)
- INFO DEFINE command [I-12](#)
- Initial object
 - DISPLAY USE OF command [D-56](#)
 - type [D-56](#)
- INITIALIZE SQL command [I-13](#)
- Initializing columns [A-33](#), [D-7](#)
- Inner joins [J-1](#), [J-2](#)
- Inner query [S-83](#), [S-84](#)
- Inoperable plan [P-22](#)
- Input host variables [D-41](#)
- INSERT operations, buffered [C-82](#)
- INSERT statement
 - access options [I-15](#)
 - and ORDER BY [S-23](#)
 - ANYWHERE clause [I-16](#)
 - APPEND clause [I-16](#)
 - description of [I-14](#)
 - exceptions [S-68](#)
 - INTO clause [I-15](#)
 - RETURNING clause [I-15](#)
 - scale of [S-46](#)
 - SYSKEY clause [I-14](#)
- INSERTABLE, in VIEWS table [V-10](#)
- Inserting
 - and check option [I-17](#)
 - and constraints [I-17](#)
 - compatible data types [I-16](#)
 - date-time values [I-17](#)
 - null values [I-17](#)
- Inserting (continued)
 - rows [I-14](#)
- Inserts, sequential [C-83](#), [C-87](#)
- INTEGER data syntax [D-5](#)
- Integrity constraint, creating [C-139](#)
- INTERACTIVE ACCESS option, CONTROL QUERY directive [C-75](#)
- Interactive interface
 - See SQLCI
- INTERVAL
 - data syntax [D-7](#)
 - data type [I-19](#)
 - literals [I-22](#)
- Interval expressions [E-22](#)
- INTERVAL values
 - in host variables [H-6](#)
 - LOAD command [L-38](#)
- INTO clause
 - FETCH statement [F-3](#)
 - INSERT statement [I-15](#)
 - SELECT statement [S-19](#)
- Invalid object [V-3](#)
- Invalid plan [P-22](#)
- Invalid program [P-22](#)
- INVALIDATE clause, CREATE INDEX statement [C-144](#)
- INVOKE directive
 - and BIND NAMES option [C-74](#)
 - and command [I-25](#)
 - NULL STRUCTURE clause [I-27](#)
- IS NULL predicate [N-6](#)
- ISLACK file attribute [I-29](#)
- ISLACK option, LOAD command [L-29](#)
- ISO
 - 8859 character sets [C-18](#)
 - character sets, ASCII [A-70](#)
 - SQL, compatibility with [S-67](#)
- Isolation Level [S-70](#)
- IXINDE01 file [C-9](#)

IXPART01

catalog table, limits [L-10](#), [L-11](#)

file [C-9](#)

IXPROG01 file [C-9](#)

IXTABL01 file [C-9](#)

IXUSAG01 file [C-9](#)

I/O (input/output)

serial or parallel writes [S-32](#)

verify disk writes [V-1](#)

J

JIS X0208 (Japanese) character set [C-18](#)

JOIN METHOD option, CONTROL TABLE directive [C-80](#)

Join queries [J-1](#)

JOIN SEQUENCE option, CONTROL TABLE directive [C-81](#)

Joins

controlling [C-86](#)

description of [J-1](#)

example of [S-30](#)

inner [J-1](#), [J-2](#)

left outer [J-1](#)

SELECT specification [S-20](#)

Julian timestamp

and COMPUTE_TIMESTAMP [C-62](#)

formatting [A-60](#)

JULIANTIMESTAMP function [J-4](#)

Justification [A-61](#), [A-63](#), [A-64](#), [A-65](#), [A-66](#)

K

Kanji character set [C-18](#)

KEY clause

CLUSTERING, CREATE TABLE statement [C-157](#)

PRIMARY, CREATE TABLE statement [C-157](#)

KEY option, LOAD command [L-21](#)

Keys

clustering [C-28](#)

Keys (continued)

column length [F-17](#)

description of [K-1](#)

displaying [F-17](#)

first, specifying value of [A-18](#), [A-37](#)

for indexes [I-9](#)

INDEX [I-9](#)

maximum values [L-13](#)

nonunique primary [C-28](#)

physical primary [P-27](#)

primary [P-27](#), [S-90](#)

system-defined primary [S-90](#)

user-defined primary [U-16](#)

KEYS catalog table [K-1](#)

Keys specifier

description of [C-145](#)

displaying [F-17](#)

KEYSEQNUMBER, in KEYS table [K-1](#)

KEYTAG clause, CREATE INDEX statement [C-145](#)

Keytags [I-9](#)

KEYTAG, in INDEXES table [I-10](#)

Key-sequenced block size [B-5](#)

Key-sequenced files [F-8](#)

Key-sequenced tables [S-90](#)

KSC5601 (Korean) character set [C-19](#)

L

Labels

for subtotal [S-88](#)

Language

displaying using ENV command [E-3](#)

elements [H-2](#)

LARGEINT data syntax [D-5](#)

Layout options

default [R-13](#)

displaying [S-50](#)

resetting [R-13](#)

saving in file [S-2](#)

setting [S-34](#)

LAYOUT option, SAVE command [S-3](#)
 LC_COLLATE section [C-32](#)
 LC_CTYPE section [C-35](#)
 LC_TDMCODESET section [C-37](#)
 Left justification [A-61](#), [A-63](#), [A-64](#), [A-65](#)
 LEFT OUTER JOIN operator [S-70](#)
 Left outer joins [J-1](#)
 LEFT_MARGIN layout option [L-1](#), [R-10](#)
 Length of page [P-2](#)
 Levels of index, displaying [F-19](#)
 Licensing [F-35](#)
 LIKE clause, CREATE TABLE statement [C-155](#)
 LIKE predicate [L-2](#)
 Limits on concurrency [C-68](#), [L-6](#)
 Lines
 folding [L-53](#)
 new line character [N-4](#)
 number on page [P-2](#)
 resetting number of [L-14](#)
 skipping in report [D-49](#)
 spacing [L-16](#)
 splitting [L-53](#)
 truncating or wrapping [S-42](#)
 LINE_NUMBER function [L-14](#), [R-11](#)
 LINE_NUMBER, default display format [L-15](#)
 LINE_SPACING layout option [L-16](#), [R-10](#)
 LIST command [L-16](#)
 LISTALL option
 DUP command [D-72](#)
 PURGE command [P-34](#)
 PURGEDATA command [P-37](#)
 SECURE command [S-9](#)
 LIST_COUNT option
 RESET SESSION command [R-19](#)
 SET SESSION command [S-41](#)
 Literals
 DATE [D-11](#)
 DATETIME [D-11](#)
 date-time [D-10](#)

Literals (continued)
 description of [L-18](#)
 INTERVAL [I-22](#)
 numeric [N-13](#)
 string [S-80](#)
 TIME [D-11](#)
 TIMESTAMP [D-11](#)
 LOAD command
 and AUDIT attribute [L-18](#), [L-34](#)
 authorization requirements [L-33](#)
 compared to APPEND [A-55](#)
 compared to COPY [L-33](#)
 conversion and loading errors [L-37](#), [L-38](#)
 description of [L-18](#)
 examples of [L-45](#)
 field conversion [L-36](#)
 field formats [L-36](#)
 operations [L-33](#)
 TMF and [L-18](#), [L-33](#)
 using tapes with [L-34](#)
 LOAD command options
 ALLOWERRORS [L-20](#)
 BLOCKIN [L-24](#)
 COMPACT [L-24](#)
 EBCDICIN [L-24](#)
 FIRST [L-21](#)
 MOVEBYNAME [L-30](#)
 MOVEBYORDER [L-31](#)
 RECIN [L-25](#)
 REDEFINE [L-31](#)
 REELS [L-26](#)
 REPLACE SPACES WITH ZEROES [L-22](#)
 REWINDIN [L-26](#)
 SHARE [L-26](#)
 SKIPIN [L-26](#)
 SOURCEDICT [L-29](#)
 SQLNULLABLE [L-23](#)
 TARGETDICT [L-30](#)

LOAD command options (continued)

TARGETREC [L-30](#)TRIM [L-26](#)TRUNC [L-31](#)UNLOADIN [L-27](#)UPSHIFT [L-23](#)VARIN [L-27](#)LOAD option, CONVERT command [C-97](#)Loading data, rules for tape use [L-34](#)Local autonomy [C-87](#)Local user [S-14](#)

LOCK TABLE statement

and CONTROL TABLE directive [L-46](#)description of [L-45](#)EXCLUSIVE clause [L-45](#)SHARE clause [L-45](#)Lock waits, CONTROL TABLE directive [C-82](#)Locking [L-47](#)

LOCKLENGTH file attribute

description of [L-51](#)displaying [F-17](#)in FILES table [F-27](#)

Locks

access by cursors [F-4](#)and FETCH [F-4](#)control system chosen [C-84](#)control timeout [C-84](#)duration [L-48](#)exclusive mode on SELECT [S-21](#)granularity [L-49](#)holder [L-50](#)in EXPLAIN report [E-17](#)limits [L-8](#)LOCK TABLE statement [L-45](#)mechanism description [L-47](#)mode [L-50](#)release summary [L-48](#)return control [C-82](#)share mode on SELECT [S-21](#)

Locks escalation

considerations [D-25](#)description of [L-49](#)LOCKS, release summary [L-48](#)

LOG command

CLEAR option [L-52](#)COMMAND option [L-52](#)description of [L-52](#)LOG file, displaying name of [E-3](#)LOGFILE [Z-12](#)Logging to a file [L-52](#)Logical lines, in reports [D-50](#)Logical names [D-27](#)Logical operators [S-5](#)Logical table [C-166](#)LOGICAL_FOLDING layout option [L-53](#), [R-10](#)**M**Magnitude [E-24](#)Manager, group [G-7](#)MANDATORY_REPORT option, SET SESSION command [S-41](#)MAP DEFINES [D-32](#), [D-33](#)

MAP NAME option

CONVERT command [C-96](#)DUP command [D-70](#)

Margins

setting left [L-1](#)setting right [R-20](#)Masks, default decimal point [D-22](#)Matching values [L-2](#)MAX function [M-1](#)MAX option, LOAD command [L-28](#)MAXEXTENTS clause, ALTER INDEX statement [A-18](#)

MAXEXTENTS file attribute

description of [M-2](#)limits [L-9](#)MAXEXTS, in FILES table [F-27](#)

MDAM option
 CONTROL QUERY directive [C-75](#)
 CONTROL TABLE directive [C-81](#)

Memory limits [L-10](#), [L-11](#)

Message file [M-3](#), [Z-17](#)
 alternate [Z-18](#)
 versions of [V-6](#)

MESSAGEFILE, displaying name of [E-3](#)

Messages
 compiler event [Z-11](#)
 displaying warning [S-42](#)
 error and warning [E-6](#)
 from SQLCOMP compiler [Z-13](#)
 non-English [Z-18](#)

MIN function [M-3](#)

Mixed views, AUDIT file attribute [A-74](#)

MODE, parameter for FastSort [Z-4](#)

Modifiers, display [A-60](#)

MODIFY CATALOG command
 REPLACE NODENAME [M-4](#)
 REPLACE VOLUME [M-11](#)

MODIFY LABEL command
 REPLACE NODENUMBER [M-19](#)
 REPLACE VOLUME [M-30](#)

MODIFY REGISTER command [M-40](#)

Modifying data [U-3](#)

Module language [S-69](#)

MOVE clause
 ALTER INDEX statement [A-16](#)
 ALTER TABLE statement [A-34](#)

MOVE options
 ALTER INDEX statement [A-16](#)

MOVEBYNAME option, LOAD command [L-30](#)

MOVEBYORDER option, LOAD command [L-31](#)

Moving
 partitions [A-34](#)
 tables [A-34](#)

Multibyte character sets [M-43](#), [S-25](#)

Multiple line heading [N-4](#)

Multivalue predicates [C-60](#)

N

NAME clause, DETAIL command [D-49](#)

NAME command [N-1](#)

NAME option [N-2](#)

Name resolution [N-2](#)

Name substitution with DEFINES [D-27](#)

Names
 alias [A-6](#)
 catalog [C-8](#)
 column [C-45](#)
 correlation [C-128](#)
 cursor [C-175](#)
 DEFINE [D-27](#)
 description of [N-3](#)
 detail alias [D-46](#)
 Guardian [G-7](#)
 OSS [O-6](#)
 partition [P-19](#)
 pathnames [O-6](#)
 subvolume [C-8](#)
 syntax and usage rules for [N-3](#)
 views [V-9](#)
 ZYQ [O-6](#)

Names buffer
 DESCRIBE statement [D-44](#)
 INCLUDE SQLDA directive [I-5](#)

National character sets [M-43](#)

National data type [M-43](#)

NATIONAL option, CONVERT command [C-99](#)

NCLUDE SQLCA Directive [I-4](#)

NEED clause
 DETAIL command [D-49](#)
 report writer option [R-11](#)

Nested subqueries [S-83](#)

Network access [S-11](#)

NEWLINE_CHAR layout option [N-4](#), [R-10](#)

NO AUDIT attribute, processing rules [T-7](#)

NO PROgid file attribute, description of [P-28](#)

NOAUDITCOMPRESS attribute, turned off for WITH SHARED ACCESS [W-6](#)

Nonaudited objects

consistency considerations for [T-9](#)

control opens [C-77](#)

FREE RESOURCES

considerations [F-30](#)

lock holder [L-50](#)

lock release summary [L-48](#)

Nonaudited tables

BUFFERED file attribute [B-11](#)

description of [N-4](#)

unlocking [U-1](#)

virtual sequential block

buffering(VSBB) [C-89](#)

NONEMPTYBLOCKCOUNT, in FILES table [F-28](#)

NonStop SMF [S-78](#)

NonStop SQL/MP

dictionary [D-1](#)

statements [S-73](#)

statements, catalog tables and [C-9](#)

version information from

FILEINFO [F-16](#)

versions of components [V-6](#)

NonStop Storage Management Foundation (SMF) [S-78](#)

Nonunique indexes [I-9](#)

NOPURGEUNTIL

ALTER CATALOG statement [A-8](#)

file attribute [N-5](#)

in FILES table [F-27](#)

NOSCRATCHON, parameter for FastSort [Z-4](#)

NOT NULL clause, CREATE TABLE statement [C-157](#)

NOT operator [S-5](#)

NOTCPUS, parameter for FastSort [Z-4](#)

NULL predicate [N-6](#)

NULL STRUCTURE clause, INVOKE directive [I-27](#)

Null values

and aggregate functions [A-77](#)

and AVG [A-77](#)

and COUNT [C-130](#)

and index sorting [C-149](#)

and INSERT [I-17](#)

and MAX [M-2](#)

and MIN [M-4](#)

and SUM [S-89](#)

and unique index [C-143](#)

copying from Enscribe [C-117](#)

defining columns [N-8](#)

description of [N-7](#)

expression evaluation [N-9](#), [N-10](#)

for Enscribe fields [L-23](#)

inserting [I-14](#)

loading from Enscribe [L-23](#)

specifying for UPDATE [U-3](#)

use in partitioned indexes [P-17](#)

NULLALLOWED in COLUMNS table [C-46](#)

NULL_DISPLAY layout option [N-10](#), [R-10](#)

Numbering lines in a report [L-14](#)

Numbers, comparing [C-59](#)

Numeric

data, comparing [C-59](#)

expressions [E-22](#)

literals [N-13](#)

NUMERIC data syntax [D-4](#)

Numeric data types

description of [N-11](#)

exact [N-11](#), [N-12](#)

O

OBEY command [O-1](#)

OBEY command files [O-1](#)

Object type, displaying [F-15](#)

Objects

- deleting damaged [C-21](#)
- dependencies [U-15](#), [V-7](#)
- determining catalog in which defined [F-16](#)
- dropping [P-33](#)
- names of [G-7](#)
- programs as [P-31](#)
- versions [V-7](#)

Objects displaying

- ownership of [D-56](#)
- security of [D-56](#)
- use of [D-54](#)

OBJECTVERSION

- in INDEXES table [I-10](#)
- in TABLES table [T-3](#)

OBJNAME, in COMMENTS table [C-51](#)OBJSUBNAME, in COMMENTS table [C-51](#)OBJTYPE, in COMMENTS table [C-51](#)OC modifier [O-10](#)OCTET_LENGTH function [O-4](#)OFFSET in COLUMNS table [C-46](#)One-way partition split, ALTER TABLE statement [A-40](#)Online dumps [W-5](#), [W-6](#)On-demand opens [C-82](#)OPEN option, CONTROL TABLE directive [C-82](#)Open state of file [F-11](#)

OPEN statement

- and SQLDA [O-5](#)
- cursors [O-5](#)
- description of [O-5](#)
- USING DESCRIPTOR clause [O-5](#)

Open states, file [F-19](#)Opening and closing tables [T-1](#)Opens, on-demand [C-82](#)Operable plan [P-22](#)Operation cost, in EXPLAIN report [E-18](#)Operations on catalog tables [C-9](#)

Operators

AND [S-5](#)Boolean or logical [S-5](#)comparison [C-60](#)NOT [S-5](#)OR [S-5](#)Optimal plan [P-22](#)

Options

CENTER_REPORT [C-12](#)COMMIT [C-52](#)DATE FORMAT [D-8](#)DECIMAL POINT [D-22](#)HEADINGS [H-2](#)LEFT_MARGIN [L-1](#)LINE_SPACING [L-16](#)LOGICAL_FOLDING [L-53](#)NAME [N-2](#)NEWLINE_CHAR [N-4](#)NULL_DISPLAY [N-10](#)OVERFLOW_CHAR [O-10](#)PAGE_COUNT [P-1](#)PAGE_LENGTH [P-2](#)REPORT [R-3](#)resetting [R-19](#)RIGHT MARGIN [R-20](#)ROLLBACK [C-55](#)ROWCOUNT [R-23](#)setting [S-45](#)SPACE [S-58](#)SUBTOTAL_LABEL [S-88](#)TIME_FORMAT [T-4](#)UNDERLINE_CHAR [U-1](#)VARCHAR_WIDTH [V-1](#)WINDOW [W-2](#)WITH SHARED ACCESS [W-4](#)OR operator [S-5](#)

ORDER BY clause

and null values [N-9](#)and UNION operator [S-26](#)

ORDER BY clause (continued)
 relation to break groups [B-8](#)
 SELECT statement [S-23](#)

ORDERING, in KEYS table [K-1](#)

ORGANIZATION clause, CREATE TABLE statement [C-158](#)

Organization of files [F-8](#)

Orphans, preventing in report [D-49](#)

OSS names [O-6](#)

OSS processes [E-2](#)

OSSFILE, in PROGRAMS table [P-32](#)

OUT command
 CLEAR option [O-7](#)
 description of [O-7](#)

OUT file
 description of [O-7](#)
 displaying name of [E-3](#)

Outer joins [J-1](#)

Outer query [S-83](#)

Outer reference [S-83](#)

Output
 device width [R-20](#), [S-42](#)
 directing to a file [O-7](#)
 host variables [D-43](#)
 lines, truncating or wrapping [S-42](#)

Output host variables [D-43](#)

OUT_REPORT command [O-8](#)

OUT_REPORT file
 current [O-9](#)
 displaying name of [E-3](#)
 specifying [O-8](#)

OVERFLOW_CHAR layout option [O-10](#), [R-10](#)

OWNER file attribute
 ALTER CATALOG statement [A-8](#)
 ALTER COLLATION statement [A-10](#)
 ALTER PROGRAM statement [A-27](#)
 ALTER TABLE statement [A-32](#)
 ALTER VIEW statement [A-50](#)
 description of [O-10](#)
 SECURE command [S-8](#)

Ownership
 displaying [D-56](#), [F-12](#)
 of a file, description of [S-14](#)

Ownership changing
 program [A-27](#)
 table [A-32](#)

Owner, generalized [G-1](#), [S-9](#), [S-14](#)

P

Packed record length [F-16](#)

PAD option
 COPY command [C-119](#)
 LOAD command [L-22](#)

Page
 advancing to next [D-49](#)
 conditional break [D-48](#)
 length [P-2](#)
 maximum in report [P-1](#)
 size for PERUSE [O-9](#)
 text at bottom of [P-1](#)

PAGE clause
 DETAIL command [D-49](#)
 report writer option [R-11](#)

PAGE FOOTING command
 AS clause [A-60](#)
 COMPUTE_TIMESTAMP function [C-62](#)
 CONCAT clause [C-63](#)
 CURRENT_TIMESTAMP function [C-174](#)
 description of [P-1](#)
 IF/THEN/ELSE clause [I-1](#)

PAGE TITLE command
 AS clause [A-60](#)
 COMPUTE_TIMESTAMP function [C-62](#)
 CONCAT clause [C-63](#)
 CURRENT_TIMESTAMP function [C-174](#)
 description of [P-4](#)

PAGE TITLE command (continued)

IF/THEN/ELSE clause [I-1](#)

PAGE_COUNT layout option [P-1](#), [R-10](#)

PAGE_LENGTH layout option [P-2](#), [R-10](#)

PAGE_NUMBER function [P-3](#), [R-11](#)

PAID (Process Access ID) [S-13](#)

Parallel execution

in EXPLAIN report [E-18](#)

loading indexes [C-145](#)

query [C-73](#), [C-74](#)

similarity, effect on [S-54](#)

PARALLEL EXECUTION clause, CREATE INDEX statement [C-145](#)

PARALLEL EXECUTION option, LOAD command [L-32](#)

Parallel index loading

description of [P-5](#)

partitions [C-145](#)

Parallel writes

compared to serial [S-32](#)

description of [S-32](#)

PARAM attribute, SAVE command [S-2](#)

Parameters

description of [P-11](#)

displaying current value [S-50](#)

dynamic SQL and default data types [P-13](#)

indicator [I-11](#)

name [P-13](#)

prepared commands [P-25](#)

resetting [R-14](#)

saving in a file [S-2](#)

setting value for [S-35](#)

specifying type of [C-4](#), [P-14](#)

unnamed [P-13](#)

using [P-13](#)

PART option, CONVERT command [C-97](#)

Partial query results [C-87](#)

PARTITION ARRAY attribute, displaying [F-20](#)

PARTITION ARRAY clause

ALTER TABLE statement [A-33](#)

CREATE TABLE statement [C-159](#)

PARTITION clause

CREATE INDEX statement [C-145](#)

CREATE TABLE statement [C-158](#)

description of [P-16](#)

Partition format [A-46](#)

PARTITIONARRAY, in FILES table [F-28](#)

PARTITIONED, in FILES table [F-27](#)

PARTITIONNAME, in PARTNS table [P-20](#)

Partitions

allocating disk space for [A-7](#)

attributes [C-108](#)

catalog description of [P-20](#)

changing array size [A-33](#)

defaults for first key columns [P-17](#)

defining [P-16](#)

description of [P-19](#)

format [P-19](#)

FORMAT file attribute [F-30](#)

in catalog [D-55](#)

in EXPLAIN report [E-16](#), [E-17](#)

loading in parallel [C-145](#)

lock for [L-48](#), [L-49](#), [L-50](#)

memory errors [L-10](#), [L-11](#)

modifying partition array [A-47](#)

moving, index [A-25](#)

names [G-7](#)

number limit [L-9](#)

performance considerations [C-134](#)

primary [P-19](#)

reusing [A-37](#), [A-47](#)

secondary [P-19](#)

size limit [L-9](#)

skip/stop at unavailable [C-83](#)

statistics for [U-10](#)

table, adding [A-40](#)

Partitions adding

index [A-20](#)table [A-35](#)

Partitions altering file attributes

index [A-15](#)table [A-32](#)

Partitions dropping

index [A-25](#)table [A-34](#), [A-47](#)

Partitions Format 2

description [F-30](#)EXTENT sizes [E-30](#)MAXEXTENTS [M-2](#)migration [A-46](#)PARTITION clause [P-16](#)

PARTNS catalog table

description of [P-20](#)limits [L-10](#), [L-11](#)PARTOF option, LOAD command [L-27](#)PARTONLY option, LOAD command [L-27](#)PARTONLYIN option, LOAD command [L-27](#)

Pascal language

and embedded SQL [E-2](#)record definition, invoking [I-26](#)Pathnames [O-6](#)Pattern matching [L-2](#)Pause while displaying rows [S-41](#)PCV (program catalog version) [V-8](#)

Performance

buffered attribute for indexes [A-39](#)catalog considerations for [C-132](#)SQL statement efficiency [E-18](#), [E-19](#), [E-20](#)

Performance considerations

catalogs [C-134](#)DDL [C-134](#)partitions [C-134](#)Peripheral Utility Program (PUP) [C-134](#)

PERUSE

command [P-21](#)page size [O-9](#)PFV (program format version) [V-8](#)Physical characteristics of object [F-9](#)

Physical primary keys

description of [P-27](#)for indexes [I-9](#)

PHYSVOL clause

CREATE INDEX statement [C-144](#)CREATE TABLE statement [C-158](#)PICTURE 9 data syntax [D-6](#)PICTURE X data syntax [D-4](#)PICTURETEXT in COLUMNS table [C-47](#)

Plans

altered [P-22](#)description of [P-22](#)inoperable [P-22](#)invalid [P-22](#)operable [P-22](#)optimal [P-22](#)valid [P-22](#)PL/1 [S-69](#)POSITION function [P-23](#)

Precision

of expression results [E-24](#)of numeric results [E-24](#)PRECISION in COLUMNS table [C-46](#)

Predicates

BETWEEN [B-4](#)comparison [C-58](#)description of [P-24](#)EXISTS [E-12](#)IN [I-3](#)in EXPLAIN report [E-16](#)IS NULL [N-6](#)LIKE [L-2](#)multivalue [C-60](#)NULL [N-6](#)quantified [Q-6](#)

- PREPARE dynamic SQL [P-5](#)
- PREPARE statement [P-25](#)
- Prepared command
 - displaying [S-51](#)
 - resetting [R-16](#)
- Primary access path [E-16](#)
- PRIMARY KEY clause, CREATE TABLE statement [C-157](#)
- Primary keys
 - catalog description of [K-1](#)
 - description of [P-27](#)
 - for indexes [I-9](#)
 - length limit [L-13](#)
 - system-defined, data types [S-90](#)
 - updateable [S-69](#)
 - user-defined [U-16](#)
- Primary partition [P-19](#)
- PRIMARYEXT2, in FILES table [F-28](#)
- PRIMARYEXT, in FILES table [F-27](#)
- PRIMARYPARTITION, in PARTNS table [P-20](#)
- Print item headings, default [D-50](#)
- Print items
 - concatenating [C-63](#)
 - conditional printing of [I-1](#)
 - default space before [S-58](#)
 - description of [P-28](#)
 - display formats [A-60](#)
 - headings [D-48](#)
 - overflow of field [O-10](#)
 - tab before printing [D-49](#)
 - width of VARCHAR text [V-1](#)
- Print list output limit
 - BREAK FOOTING command [B-6](#)
 - BREAK TITLE command [B-10](#)
 - DETAIL command [D-50](#)
 - PAGE FOOTING command [P-2](#)
 - PAGE TITLE command [P-4](#)
 - REPORT FOOTING command [R-2](#)
 - REPORT TITLE command [R-6](#)
- Print lists [P-28](#)
- Printer, device width [R-20](#)
- PRI, parameter for FastSort [Z-4](#)
- Process
 - alter owner [A-27](#), [A-28](#)
 - device width used [R-20](#)
- Process Access ID (PAID) [S-13](#)
- Process File Segment (PFS), limits [L-10](#), [L-11](#)
- Process File Segment(PFS) and DEFINES [D-27](#)
- increasing [L-10](#)
- memory requirements [L-10](#), [L-11](#)
- Processing
 - concepts for TMF transactions [T-6](#)
 - control locking [C-77](#)
 - control parallel execution [C-73](#), [C-74](#)
 - control query processing [C-74](#)
 - control table opens [C-77](#)
- Processing rules
 - audited objects [T-7](#)
 - DDL and DML statements [T-7](#)
 - NO AUDIT attribute [T-7](#)
 - SQLCI [T-8](#)
- PROCESSNAME, in TRANSIDS table [T-11](#)
- PROGID attributes, in PROGRAMS table [P-31](#)
- PROGID attribute, alter owner [A-27](#), [A-28](#)
- PROGID file attribute
 - and security [S-13](#)
 - description of [P-28](#)
 - SECURE command [S-9](#)
- Program catalog version (PCV) [V-8](#)
- Program execution
 - authority for [S-15](#)
 - cost, in EXPLAIN report [E-18](#), [E-19](#), [E-20](#)
 - displaying cost [D-52](#)
 - dynamic [E-11](#)
 - in EXPLAIN report [E-14](#), [E-18](#)
- Program file, validation [P-28](#)

Program format version (PFV) [V-8](#)
 PROGRAM INVALIDATION [P-28](#)
 PROGRAMCATALOGVERSION, in PROGRAMS table [P-32](#)
 PROGRAMFORMATVERSION, in PROGRAMS table [P-32](#)
 Programmatic SQL [E-2](#)
 Programming language, description of [E-2](#)
 PROGRAMNAME, in PROGRAMS table [P-31](#)
 Programs
 ALTER PROGRAM statement [A-27](#)
 altering security attributes [A-27](#)
 catalog description of object in [P-31](#)
 compiling from Guardian [S-67](#)
 compiling from OSS [C-176](#)
 dependent, effect of CLEANUP [C-23](#)
 DROP statement [D-65](#)
 dropping [D-65](#), [P-33](#)
 duplicating [D-69](#), [D-75](#)
 file recompilation [P-28](#)
 HOSV [V-8](#)
 invalid [P-22](#)
 names [G-7](#)
 ownership changes [A-27](#)
 PCV [V-8](#)
 PFV [V-8](#)
 purging [D-64](#)
 renaming [A-27](#)
 Safeguard protection [A-28](#)
 security changes [A-27](#)
 using DEFINES from [D-31](#)
 versions of [V-8](#)
 PROGRAMS catalog table
 and program invalidation [P-28](#), [P-31](#)
 description of [P-31](#)
 PROGRAM, parameter for FastSort [Z-4](#)
 Prompts [S-60](#)

Protection views
 AUDIT file attribute [A-74](#)
 description of [P-32](#), [V-9](#)
 limit per table [L-14](#)
 similarity [S-55](#)
 PROTECTION, in VIEWS table [V-10](#)
 PS Text Edit [T-3](#)
 PUP utility [U-17](#)
 PUP (Peripheral Utility Program) [C-134](#)
 Purge
 access [S-14](#)
 date and time, changing [A-32](#)
 objects [D-63](#)
 PURGE command
 and TMF [P-34](#), [P-35](#)
 description of [P-33](#)
 LISTALL option [P-34](#)
 SHADOWSONLY option [P-34](#)
 PURGEDATA command
 and TMF [P-37](#), [P-38](#)
 description of [P-36](#)
 LISTALL option [P-37](#)

Q

Qualified column names [C-46](#)
 Qualified fileset list [Q-1](#)
 Quantified predicate [Q-6](#)
 Queries
 inner [J-1](#)
 join [J-1](#)
 outer [J-1](#)
 Query execution plans [P-22](#)
 Query optimization
 CONTROL EXECUTOR directive [C-73](#)
 CONTROL QUERY directive [C-75](#)
 equijoin for parallel execution [C-73](#)
 parallel processing [C-73](#), [C-74](#)
 using current statistics [S-78](#)

Query processing

getting partial results [C-87](#)table reference limit [L-13](#)Quotation marks within string literals [S-81](#)**R**Random access block size [B-5](#)Range of values, selecting [B-4](#)Read access [S-14](#)READ COMMITTED [S-70](#)READ operations, buffered [C-82](#)READ UNCOMMITTED [S-70](#)

Reading

data [S-17](#)row [F-3](#)REAL data syntax [D-5](#)RECIN option, LOAD command [L-25](#)RECLength file attribute [R-1](#)RECOMPILEMODE, in PROGRAMS table [P-32](#)RECOMPILETIME, in PROGRAMS table [P-31](#)

Record

definition, invoking [I-26](#)descriptions, catalog tables [B-1](#)descriptions, tables and views [I-25](#)nonconvertible [C-116](#), [L-20](#)

Record length

displaying [F-16](#)RECLength file attribute [R-1](#)RECORDSIZE, in FILES table [F-28](#)RECOU option, COPY command [C-119](#)

REDEFINE option

CONVERT command [C-98](#)LOAD command [L-31](#)Redefinition time [F-19](#)REDEFTIME, in TABLES table [T-2](#)REELS option, LOAD command [L-26](#)

REFERENCES

column constraint [S-68](#)column privileges [S-68](#)REGISTERONLY, in PROGRAMS table [P-32](#)RELATIONSHIPTYPE, in USAGES [U-15](#)Relative files [F-8](#)

Relative tables

and SYSKEYS [S-90](#)file attribute summary [F-7](#)organization [F-8](#)reserved row length [R-1](#)system key data type [S-90](#)RELEASE statement [R-1](#)Releases [V-6](#)Remote objects and CLEANUP [C-24](#)Remote user [S-14](#)

RENAME clause

ALTER COLLATION statement [A-10](#)ALTER INDEX statement [A-15](#)ALTER PROGRAM statement [A-27](#)ALTER TABLE statement [A-32](#)ALTER VIEW statement [A-50](#)

Renaming

collations [A-10](#)program [A-27](#)tables [A-32](#)views [A-50](#)REPAIR option [V-3](#)Repairing Views [V-5](#)

REPEATABLE access

description of [A-2](#)in EXPLAIN report [E-16](#)nonaudited tables [A-2](#)REPEATABLE READ [S-70](#)Repeating command [F-1](#), [O-1](#), [Z-1](#)

REPLACE SPACES WITH option

COPY command [C-116](#)LOAD command [L-22](#)REPORT [R-2](#)

Report

- cancelling [C-2](#)
- centering contents of [C-12](#)
- copies [O-9](#)
- default format [D-50](#)
- default right margin [L-53](#)
- file, closing [O-8](#)
- file, specifying [O-8](#)
- file, specifying or closing [O-8](#)
- form feed for [P-2](#)
- form name [O-9](#)
- headings [D-50](#)
- line length [D-50](#)
- logical lines [D-50](#)
- naming [O-9](#)
- number of copies, specifying [O-9](#)
- number of rows before pause [S-41](#)
- numbering lines [L-14](#)
- page size for PERUSE [O-9](#)
- preventing orphans [D-49](#)
- preventing widows [D-49](#)
- summaries [D-50](#)
- width [L-53](#)

REPORT FOOTING command

- AS clause [A-60](#)
- COMPUTE_TIMESTAMP function [C-62](#)
- CONCAT clause [C-63](#)
- CURRENT_TIMESTAMP function [C-174](#)
- description of [R-2](#)
- IF/THEN/ELSE clause [I-1](#)

Report formatting commands

- deleting stored [R-16](#)
- saving in file [S-2](#)

Report layout options

- CENTER_REPORT [C-12](#), [R-10](#)
- DATE_FORMAT [D-8](#), [R-10](#)
- DECIMAL_POINT [D-22](#), [R-10](#)
- HEADINGS [H-2](#), [R-10](#)

Report layout options (continued)

- LEFT_MARGIN [L-1](#), [R-10](#)
- LINE_SPACING [L-16](#), [R-10](#)
- LOGICAL_FOLDING [L-53](#), [R-10](#)
- NEWLINE_CHAR [N-4](#), [R-10](#)
- NULL_DISPLAY [N-10](#), [R-10](#)
- OVERFLOW_CHAR [O-10](#), [R-10](#)
- PAGE_COUNT [P-1](#), [R-10](#)
- PAGE_LENGTH [P-2](#), [R-10](#)
- RIGHT_MARGIN [R-10](#), [R-20](#)
- ROWCOUNT [R-10](#), [R-23](#)
- SPACE [R-10](#), [S-58](#)
- SUBTOTAL_LABEL [R-10](#), [S-88](#)
- TIME_FORMAT [R-10](#), [T-4](#)
- UNDERLINE_CHAR [R-10](#), [U-1](#)
- VARCHAR_WIDTH [R-10](#), [V-1](#)
- WINDOW [R-10](#), [W-2](#)

REPORT option, SAVE command [S-3](#)

REPORT TITLE command

- AS clause [A-60](#)
- COMPUTE_TIMESTAMP function [C-62](#)
- CONCAT clause [C-63](#)
- CURRENT_TIMESTAMP function [C-174](#)
- description of [R-6](#)
- IF/THEN/ELSE clause [I-1](#)

Report writer [R-7](#)

Report writer clauses

- AS [A-60](#)
- AS DATE/TIME [A-67](#)
- CONCAT [C-63](#)
- IF/THEN/ELSE [I-1](#)
- NEED [D-49](#)
- PAGE [D-49](#)
- SKIP [D-49](#)
- SPACE [D-49](#)
- TAB [D-49](#)

Report writer functions

COMPUTE_TIMESTAMP [C-62](#)CURRENT_TIMESTAMP [C-174](#)LINE_NUMBER [L-14](#)

Reports

form feeds [P-3](#)summary [D-50](#)suppressing headings [H-1](#)Reserved words [R-11](#)RESET DEFINE command [R-12](#)RESET LAYOUT command [R-13](#)RESET PARAM command [R-14](#)RESET PREPARED command [R-16](#)RESET REPORT command [R-16](#)RESET SESSION command [R-19](#)RESET STYLE command [R-19](#)RESETBROKEN clause, ALTER INDEX statement [A-18](#)RESETBROKEN file attribute [R-20](#)RESTORE utility [U-17](#)Return control of locks [C-82](#)

RETURN IF LOCKED option

and converted Enscribe applications [C-82](#)CONTROL TABLE directive [C-82](#)

RETURNING clause

EXECUTE statement [E-9](#)INSERT statement [I-15](#)REUSE PARTITION clause, ALTER TABLE statement [A-37](#)

Reusing partitions

description of [A-47](#)table [A-37](#)REVOKE statement, ANSI/ISO SQL [S-68](#)REWINDIN option, LOAD command [L-26](#)REWINDOUT, COPY command [C-120](#)Right justification [A-61](#), [A-63](#), [A-64](#), [A-65](#)RIGHT_MARGIN layout option [R-10](#), [R-20](#)ROLLBACK option [C-55](#)ROLLBACK WORK statement [R-21](#)Row count display, suppressing [R-23](#)ROWCOUNT layout option [R-10](#), [R-23](#)ROWCOUNT, in BASETABS table [B-1](#)

Rows

deleting [D-39](#)displaying definition of [I-25](#)displaying or printing [L-16](#)displaying subset of [W-3](#)fetching [F-3](#)inserting [I-14](#)length limit [L-12](#)pausing while displaying [S-41](#)RECLENGTH file attribute [R-1](#)report format for [D-47](#)selecting [S-17](#)specifying number to display [S-41](#)updating [U-3](#)

ROWSIZE

in BASETABS table [B-1](#)in INDEXES table [I-10](#)RPTSQL [R-3](#)rwep security string [S-14](#)**S**

Safeguard

program file security [A-28](#)security system [S-11](#), [S-13](#), [S-68](#)Sample database [S-1](#)

SAVE command

ALL option [S-2](#)COMMAND option [S-3](#)DEFINES option [S-2](#)description of [S-2](#)ENV option [S-2](#)LAYOUT option [S-3](#)PARAM option [S-2](#)REPORT option [S-3](#)section header [S-4](#)SESSION option [S-3](#)STYLE option [S-3](#)

SAVEALL option, DUP command [D-73](#)
 Saved commands, executing [O-1](#)
 SAVEID option, DUP command [D-73](#)
 Saving commands in a file [S-2](#)
 SBB, in EXPLAIN report [E-19](#)
 Scale
 of expression results [E-24](#)
 of numeric results [E-24](#)
 SETSCALE function [S-46](#)
 SCALE in COLUMNS table [C-46](#)
 Scientific notation [N-13](#)
 SCRATCH option, LOAD command [L-28](#)
 SCRATCHON, parameter for FastSort [Z-4](#)
 SCRATCH, parameter for FastSort [Z-4](#)
 Search conditions
 description of [S-5](#)
 table selectivity [E-19](#)
 WHERE clauses [W-2](#)
 SEARCH DEFINES [D-32](#)
 Secondary partition [P-19](#)
 SECONDARYEXT2, in FILES table [F-28](#)
 SECONDARYEXT, in FILES table [F-27](#)
 SECONDHIGHVALUE in COLUMNS table [C-46](#)
 SECONDLOWVALUE in COLUMNS table [C-46](#)
 Section header
 in OBEY file [O-1](#)
 using with SAVE command [S-4](#)
 Section names, in command files [O-2](#)
 SECURE clause
 CREATE CATALOG statement [C-132](#)
 CREATE TABLE statement [C-159](#)
 CREATE VIEW statement [C-168](#)
 SECURE command
 ALLOWERRORS option [S-8](#)
 and REVOKE [S-68](#)
 CLEARONPURGE option [S-8](#)
 description of [S-7](#)
 LISTALL option [S-9](#)
 OWNER option [S-8](#)

SECURE command (continued)
 PROGID option [S-9](#)
 TMF and [S-9](#)
 SECURE file attribute
 ALTER CATALOG statement [A-8](#)
 ALTER COLLATION statement [A-10](#)
 ALTER INDEX statement [A-15](#)
 ALTER PROGRAM statement [A-27](#)
 ALTER TABLE statement [A-32](#)
 ALTER VIEW statement [A-50](#)
 description of [S-11](#)
 Security
 catalogs and [A-9](#)
 description of [S-11](#)
 displaying [D-56](#), [F-12](#)
 Safeguard subsystem and [A-28](#)
 specifying view [C-168](#)
 super ID [S-13](#)
 table and view dependencies and [A-38](#)
 Security changing
 index [A-15](#)
 program [A-27](#)
 table [A-32](#)
 view [A-50](#)
 Security file attributes for
 CLEARONPURGE [C-26](#)
 NOPURGEUNTIL [N-5](#)
 OWNER [O-10](#)
 SECURE [S-11](#)
 VERIFIEDWRITES [V-1](#)
 Security strings [S-14](#)
 Security string, file attribute setting [S-11](#)
 SECURITYMODE
 in INDEXES table [I-10](#)
 in PROGRAMS table [P-31](#)
 in TABLES table [T-3](#)
 SECURITYVECTOR
 in INDEXES table [I-10](#)
 in PROGRAMS table [P-31](#)
 in TABLES table [T-2](#)

- SEGMENT, parameter for FastSort [Z-4](#)
- SELECT command, directing output of [O-8](#)
- SELECT DISTINCT clause, and CREATE VIEW statement [S-68](#)
- SELECT statement
 - and scale [S-47](#)
 - BROWSE access [S-21](#)
 - cancelling, in SQLCI [C-2](#)
 - description of [S-18](#)
 - DISTINCT clause [D-57](#), [S-19](#)
 - dynamic SQL [S-24](#)
 - FOR UPDATE OF clause [S-24](#)
 - FROM clause [S-20](#)
 - GROUP BY clause [S-22](#)
 - HAVING clause [S-21](#)
 - INTO clause [S-19](#)
 - lock mode option [S-21](#)
 - lock release summary [L-48](#)
 - lock release summary [L-48](#)
 - ORDER BY clause [S-23](#)
 - restrictions for view definitions [C-167](#)
 - suppressing row count display [R-23](#)
 - table privileges [S-68](#)
 - UNION operator [S-23](#)
- Selecting data
 - cursor declaration [D-23](#), [D-24](#)
 - distinct rows [S-19](#)
 - matching values [L-2](#)
 - search conditions [S-5](#)
 - value comparisons [C-60](#)
 - values from a list [B-4](#)
 - values to insert [I-14](#)
 - values within a range [B-4](#)
- Selecting from catalog tables [S-30](#)
- Selecting rows
 - for DELETE operation [D-39](#)
 - for INSERT operation [I-14](#)
 - for UPDATE operation [U-3](#)
- Select-in-progress prompt [S-60](#)
- Select-list
 - dependencies on GROUP BY [S-25](#)
 - for distinct rows [S-19](#)
- Self join [S-30](#)
- SEQNUMBER
 - in COMMENTS table [C-51](#)
 - in CONSTRNT table [C-70](#)
 - in CPRLSRCE table [C-131](#)
- Sequential access block size [B-5](#)
- SEQUENTIAL BLOCKSPLIT option, CONTROL TABLE directive [C-83](#)
- Sequential blocksplits [E-19](#)
- Sequential cache [E-16](#)
- Sequential inserts [C-83](#)
- SEQUENTIAL option, CONTROL TABLE directive [C-82](#)
- Serial writes
 - compared to parallel [S-32](#)
 - description of [S-32](#)
- SERIALIZABLE, ISO Isolation Level [S-70](#)
- SERIALWRITES file attribute [S-32](#)
- SERIALWRITES, in FILES table [F-27](#)
- Session
 - ending [E-13](#)
- Session options
 - displaying [S-53](#)
 - resetting [R-19](#)
 - saving in file [S-2](#)
 - setting [S-39](#)
- SESSION option, SAVE command [S-3](#)
- SET DEFINE command [S-32](#)
- SET DEFMODE command [S-34](#)
- SET LAYOUT command [S-34](#)
- Set operation
 - DELETE statement [D-39](#)
 - UPDATE statement [U-3](#)
- SET PARAM command
 - COMPUTE_TIMESTAMP function [C-62](#)
 - CURRENT_TIMESTAMP function [C-174](#), [S-35](#)

- SET PARAM command (continued)
 - description of [S-35](#)
- SET SESSION command
 - AUTOWORK option [S-39](#)
 - BREAK_KEY option [S-40](#)
 - description of [S-39](#)
 - DISPLAY_ERROR option [S-40](#)
 - ERROR_ABORT option [S-40](#)
 - ERROR_TEXT option [S-41](#)
 - LIST_COUNT option [S-41](#)
 - MANDATORY_REPORT option [S-41](#)
 - STATISTICS option [S-41](#)
 - WARNINGS option [S-42](#)
 - WRAP option [S-42](#)
- SET STYLE command [S-45](#)
- SETMODE 91,3 option [C-83](#)
- SETSCALE function [S-46](#)
- shadow label
 - and CLEANUP [C-22](#)
 - and FILEINFO [F-10](#)
 - description of [F-12](#)
- Shadow labels
 - and CLEANUP command [C-22](#)
 - and FILEINFO command [F-10](#)
 - description of [F-12](#)
 - purging [P-33](#)
- SHADOWONLY option, PURGE command [P-34](#)
- SHARE clause, LOCK TABLE statement [L-45](#)
- SHARE option, LOAD command [L-26](#)
- SHARED ACCESS on DDL statements [W-4](#)
- Shared lock
 - description of [L-50](#)
 - on SELECT statement [S-21](#)
 - SHARE clause [L-45](#)
- SHAREOUT option, COPY command [C-122](#)
- Shift JIS character set [C-18](#)
- Shorthand views
 - and UNION operator [C-170](#)
 - AUDIT file attribute [A-74](#)
 - based on joins [C-170](#)
 - description of [S-48](#), [V-9](#)
- SHOW CONTROL command [S-48](#)
- SHOW DEFINE command [S-48](#)
- SHOW DEFMODE command [S-49](#)
- SHOW LAYOUT command [S-50](#)
- SHOW PARAM command [S-50](#)
- SHOW PREPARED command [S-51](#)
- SHOW REPORT command [S-52](#)
- SHOW SESSION command [S-53](#)
- SHOW STYLE command [S-54](#)
- Similarity between protection views [S-55](#)
- Similarity between tables [S-55](#)
- SIMILARITY CHECK clause
 - ALTER TABLE statement [A-32](#)
 - ALTER VIEW statement [A-51](#)
 - CREATE TABLE statement [C-160](#)
 - CREATE VIEW statement [C-168](#)
- Similarity checks [S-54](#)
- SIMILARITYCHECK, in TABLES table [T-3](#)
- SIMILARITYINFO, in PROGRAMS table [P-32](#)
- Simple file names [G-7](#)
- Simple fileset list [F-29](#)
- Single spacing [L-16](#)
- SKIP clause
 - DETAIL command [D-49](#)
 - report writer option [R-11](#)
- SKIP option, CONTROL TABLE directive [C-83](#)
- SKIPIN option, LOAD command [L-26](#)
- SKIPOUT option, COPY command [C-120](#)
- SLACK file attribute [S-57](#)
- SLACK option, LOAD command [L-29](#)
- SMALLINT data syntax [D-5](#)
- SMF (Storage Management Foundation) [S-78](#)
- SOME, in quantified predicate [Q-6](#)

- [SORT DEFINES](#) [D-32](#)
- [SORTED](#) option, [LOAD](#) command [L-27](#)
- Sorting
 - [ORDER BY](#) clause on [SELECT](#) [S-23](#)
 - parameters for [Z-4](#)
 - performance [Z-4](#)
- [SORTPROG](#) process, controlling [S-25](#)
- [SOURCE](#) option, [CONVERT](#) command [C-97](#)
- [SOURCEDATE](#) option, [DUP](#) command [D-73](#)
- [SOURCEDICT](#) option, [LOAD](#) command [L-29](#)
- [SOURCEREC](#) option, [LOAD](#) command [L-29](#)
- Space available in file [F-19](#)
- [SPACE](#) clause, [DETAIL](#) command [D-49](#)
- [SPACE](#) clause, report writer option [R-11](#)
- [SPACE](#) layout option [R-10](#), [S-58](#)
- Spaces, suppressing in printing [C-63](#)
- Splitting
 - indexes [A-19](#)
 - partitions, [ALTER TABLE](#) statement [A-34](#)
 - tables [A-34](#)
 - the default detail line [L-53](#)
- [SPOOL DEFINES](#) [D-33](#)
- Spooler
 - device width [R-20](#)
 - use of [O-8](#)
- SQL
 - and host communication [H-6](#)
 - compilation, prepare statements [P-25](#)
 - compiler event messages [Z-11](#)
 - identifiers [S-59](#)
 - message file [Z-18](#)
 - programmatic [E-2](#)
 - statement execution time [D-52](#)
 - statements [S-73](#)
 - static [S-77](#)
- SQL compiler event messages [Z-11](#)
- SQL directive [S-59](#)
- SQL SENSITIVE flag [F-35](#)
- SQL tables
 - converting an Enscribe file to [C-94](#)
 - loading data into [L-18](#)
 - rules for copying [C-124](#)
- SQL VALID flag [F-35](#)
- SQLCA
 - description of [I-4](#)
 - effect of [DECLARE CURSOR](#) [D-24](#)
- SQLCI
 - commands [S-63](#)
 - description of [S-60](#)
 - processing rules [T-8](#)
 - prompts [S-60](#)
 - summary of report writer commands [R-7](#)
 - using [DEFINES](#) from [D-30](#)
- SQLCODE [S-67](#)
- SQLCOMP command
 - authority for [S-15](#)
 - description of [S-67](#)
 - with [CATALOG DEFINE](#) [D-37](#)
 - with [DEFINES](#) [D-31](#)
- SQLDA
 - and [FETCH](#) statement [F-3](#)
 - and [OPEN](#) statement [O-5](#)
 - [DESCRIBE INPUT](#) statement [D-41](#)
 - description of [I-4](#)
 - [INCLUDE SQLDA](#) directive [I-4](#)
- SQLNULLABLE option
 - [COPY](#) command [C-117](#)
 - [LOAD](#) command [L-23](#)
 - load operations [L-40](#)
- SQLSA
 - description of [I-6](#)
 - effect of [DECLARE CURSOR](#) [D-24](#)
 - [INCLUDE SQLSA](#) directive [I-6](#)
- SQLSTATE status codes [S-69](#)

SQL.CATALOGS table

See also CATALOGS table
 and CREATE CATALOG [C-134](#)
 and CREATE SYSTEM CATALOG
 command [C-152](#)
 description of [C-11](#)

STABLE access

concurrency summary [A-3](#)
 description of [A-1](#)
 in EXPLAIN report [E-16](#)
 nonaudited tables [A-2](#)
 UPDATE statement [U-3](#)

Standard prompt [S-60](#)Standards conformance [S-67](#)Starting to log to a file [L-52](#)

Statements

ALTER CATALOG [A-8](#)
 ALTER COLLATION [A-10](#)
 ALTER INDEX [A-13](#)
 ALTER PROGRAM [A-27](#)
 ALTER TABLE [A-29](#)
 ALTER TABLE ADD
 CONSTRAINT [S-70](#)
 ALTER TABLE DROP
 CONSTRAINT [S-70](#)
 ALTER VIEW [A-50](#)
 atomicity [S-68](#)
 BEGIN WORK [B-3](#)
 CLOSE [C-26](#)
 COMMENT [C-48](#)
 COMMIT WORK [C-56](#)
 CONTINUE [C-70](#)
 cost of executing [D-52](#)
 CREATE ASSERTION (ANSI/ISO
 SQL) [S-71](#)
 CREATE CATALOG [C-132](#)
 CREATE COLLATION [C-137](#)
 CREATE CONSTRAINT [C-139](#)
 CREATE INDEX [C-142](#)
 CREATE TABLE [C-154](#)

Statements (continued)

CREATE VIEW [C-166](#)
 DCL [D-18](#)
 DDL [D-20](#)
 DECLARE CURSOR [D-23](#)
 DELETE [D-39](#)
 DESCRIBE [D-43](#)
 DESCRIBE INPUT [D-41](#)
 description of [S-73](#)
 DROP [D-63](#)
 DROP ASSERTION [S-71](#)
 DSL [D-67](#)
 EXECUTE [E-7](#)
 EXECUTE IMMEDIATE [E-11](#)
 FETCH [F-3](#)
 FREE RESOURCES [F-30](#)
 GET CATALOG OF SYSTEM [G-1](#)
 GET VERSION [G-2](#)
 GET VERSION OF PROGRAM [G-4](#)
 HELP TEXT [H-4](#)
 INSERT [I-14](#)
 LOCK TABLE [L-45](#)
 maximum length [L-13](#)
 name of [P-25](#)
 OPEN [O-5](#)
 PREPARE [P-25](#)
 prepared [P-25](#)
 RELEASE [R-1](#)
 ROLLBACK WORK [R-21](#)
 SELECT [S-18](#)
 summary [S-73](#)
 UNLOCK TABLE [U-1](#)
 UPDATE [U-3](#)
 UPDATE STATISTICS [U-7](#)
 use with catalog tables [C-9](#)

Static SQL [S-77](#)

Statistics

description of [S-78](#)
 displaying [D-52](#)

Statistics (continued)

- enabling display of [S-41](#)
- file [F-22](#)
- for partition [U-10](#)
- from PREPARE [P-25](#)
- retrieving from catalog tables [U-10](#)
- selecting current compiler with [U-9](#)
- updating [U-7](#)
- using for better access plan [S-78](#)

STATISTICS display, FILEINFO command [F-22](#)

STATISTICS option

- FILEINFO command [F-10](#)
- SET SESSION command [S-41](#)

STATISTICSTIME, in BASETABS table [B-1](#)

Status, checking with SQLCA [I-4](#)

STOP AT option, CONTROL TABLE directive [C-83](#)

Stopping a SELECT command [C-2](#)

Storage Management Foundation (SMF) [S-78](#)

Stored commands

- deleting [R-16](#)
- displaying [S-52](#)

stored Commands

- displaying [S-52](#)

stored commands

- deleting [R-16](#)
- displaying [S-52](#)

String functions [S-80](#)

String literals

- and quotation marks [S-81](#)
- description of [S-80](#)

Structured files [F-8](#)

Style options

- displaying [S-54](#)
- resetting [R-19](#)
- saving in file [S-2](#)
- setting [S-45](#)
- UNDERLINE_CHAR [U-1](#)

STYLE option, SAVE command [S-3](#)

Subqueries

- comparing expression to results [Q-6](#)
- correlated [S-83](#)
- description of [S-82](#)
- in EXISTS predicate [E-12](#)
- nested [S-83](#)
- nesting limit of [L-13](#)

SUBSORT DEFINES [D-33](#)

Subsorts for loading indexes [C-145](#)

SUBSORTS, parameter for FastSort [Z-4](#)

Substituting values with parameters [P-11](#)

SUBSTRING function [S-84](#)

SUBSYSTEMNAME

- in CATALOGS table [C-11](#)
- in VERSIONS table [V-9](#)

SUBTOTAL command

- and BREAK ON command [S-86](#)
- description of [S-86](#)

SUBTOTAL_LABEL layout option [R-10](#), [S-88](#)

Subvolumes

- default [V-11](#)
- names of [C-8](#)

SUM function [S-89](#)

Summary reports [D-50](#)

Super ID

- description of [S-90](#)
- used by audit fix-up process [W-6](#)

SUPER.SUPER user [S-13](#)

Suppressing headings in reports [H-1](#)

Suppressing row count display [R-23](#)

SWAP, parameter for FastSort [Z-4](#)

SYNCDEPTH option, CONTROL TABLE directive [C-84](#)

SYSKEY

- and user-defined keys [U-16](#)
- column [C-28](#), [S-90](#)

SYSKEY clause, INSERT statement [I-14](#)

SYSKEYs [S-90](#)

SYSNUMBER, in TRANSIDS table [T-11](#)

System catalog (continued)

creating [C-152](#)description of [S-92](#)dropping [D-66](#)upgrading [U-13](#)versions [V-7](#)SYSTEM command [S-93](#)System default multibyte character set [M-43](#)SYSTEM default value [D-27](#)

System defines

summary [S-94](#)=_AUDSERV_XSWAP_node [Z-2](#)=_DEFAULTS [Z-3](#)=_SORT_DEFAULTS [Z-4](#)=_SQL_AUD_node [Z-14](#)=_SQL_CAT_HEAP_LIMIT [Z-6](#)=_SQL_CAT_node [Z-14](#)=_SQL_CI2_node [Z-14](#)=_SQL_CMP_CPUS_node [Z-7](#)=_SQL_CMP_DOUBLE_SBB_OFF [Z-8](#)=_SQL_CMP_DOUBLE_SBB_ON [Z-9](#)=_SQL_CMP_EQ_LIMIT [Z-10](#)=_SQL_CMP_EVENT [Z-11](#)=_SQL_CMP_EVENT_NO0 [Z-12](#)=_SQL_CMP_node [Z-14](#)=_SQL_cmp_node [Z-14](#)=_SQL_CMP_NO_KS_MJOIN [Z-14](#)=_SQL_EXE_DOUBLE_SHUTOFF [Z-15](#)=_SQL_EXE_ESPS_CK_CMON [Z-16](#)=_SQL_EXE_USE_SWAPVOL [Z-17](#)=_SQL_MSG_node [Z-18](#)=_SQL_RECGEN_node [Z-20](#)=_SQL_TM_node_vol [Z-20](#)=_SQL_UTL_node [Z-14](#)SYSTEM, default [S-93](#)System, default [E-3](#)

System-defined primary key

description of [S-90](#)maximum values [L-13](#)returned value on INSERT [I-17](#)S> prompt [S-60](#)

T

TAB clause

DETAIL command [D-49](#)report writer option [R-11](#)use with report window [W-2](#)

Table

BASETABS catalog table [B-1](#)CATALOGS catalog table [C-11](#)CPRLSRCE catalog table [C-131](#)CPRULES catalog table [C-131](#)FILES catalog table [F-27](#)INDEXES catalog table [I-10](#)PARTNS catalog table [P-20](#)PROGRAMS catalog table [P-31](#)TABLES catalog table [T-2](#)TRANSIDS catalog table [T-11](#)USAGES catalog table [U-15](#)VERSIONS catalog table [V-9](#)VIEWS catalog table [V-10](#)

TABLECODE

in INDEXES table [I-10](#)in TABLES table [T-2](#)TABLECODE file attribute [T-1](#)TABLECOLNUMBER, in KEYS table [K-1](#)TABLELOCK option, CONTROL TABLE directive [C-84](#)

TABLENAME

in BASETABS table [B-1](#)in COLUMNS table [C-46](#)in CONSTRNT table [C-70](#)in INDEXES table [I-10](#)in TABLES table [T-2](#)

Tables

address of EOF [F-19](#)
 altering attributes [A-32](#)
 attribute specification [C-154](#)
 available space [F-19](#)
 base [T-1](#)
 catalog [C-8](#)
 catalog description of [T-2](#)
 clearing data from [P-36](#)
 column limit [L-6](#)
 COLUMNS [C-46](#)
 COMMENTS [C-51](#)
 CONSTRNT [C-70](#)
 copying to an Enscribe file [C-113](#)
 CREATE TABLE statement [C-154](#)
 date-caused program
 recompilation [F-19](#)
 description of [T-1](#)
 displaying [C-123](#)
 DROP statement [D-65](#)
 dropping [P-33](#)
 duplicating [D-69](#), [D-74](#)
 expiration date, setting [N-5](#)
 file statistics [F-22](#)
 fragmented, updating statistics [U-9](#)
 index dependencies [C-148](#)
 inserting with check option [I-17](#)
 joining [S-30](#)
 limit on tables per query [L-13](#)
 loading data into [L-18](#)
 locking [L-45](#), [L-50](#)
 moving [A-34](#)
 names [G-7](#)
 organizations [F-8](#)
 ownership change [A-32](#)
 partitioned [P-19](#)
 partitioning [L-12](#)
 physical file attributes [C-108](#)
 purging [D-64](#)

Tables (continued)

renaming [A-32](#)
 row definition [I-25](#)
 security change [S-7](#)
 selecting from [S-17](#)
 selectivity [E-19](#)
 setting organization for [C-158](#)
 similarity rules for [S-55](#)
 specifying file attributes of [C-160](#)
 temporary [T-4](#), [Z-17](#)
 transaction limit [L-13](#)
 unlocking nonaudited [U-1](#)
 unpartitioned [P-19](#)
 updating rows of [U-3](#)
 updating statistics for [U-7](#)
 using ALTER TABLE statement [A-29](#)
 versions [V-7](#)
 view, limit [L-6](#), [L-8](#), [L-13](#)
 view, security dependencies [A-38](#)
 TABLES catalog table [T-2](#)
 Tables partitions
 adding [A-35](#)
 dropping [A-34](#)
 TABLETYPE, in TABLES table [T-2](#)
 TAL language
 and embedded SQL [E-2](#)
 record definition, invoking [I-26](#)
 TAPE DEFINES [D-33](#)
 Tapes, using with LOAD command [L-34](#)
 TARGET option, DUP command [D-73](#)
 TARGETDICT option, LOAD
 command [L-30](#)
 TARGETREC option, LOAD
 command [L-30](#)
 TEDIT command [T-3](#)
 temporal scoping [C-76](#)
 Temporary files, size limit [L-13](#)
 Temporary tables
 and DEFINE [Z-17](#)
 description of [T-4](#)

- Terminal, device width [R-20](#)
- Terminating a SELECT command [C-2](#)
- Testing for errors [W-1](#)
- Text editor [T-3](#)
- Text, display format [A-60](#)
- TEXT, in CPRLSRCE table [C-131](#)
- The [M-3](#)
- Time
 - compilation and execution of [D-52](#)
 - computing for report [C-62](#), [C-173](#)
 - of expiration [A-32](#)
 - print item display format [A-67](#)
- TIME data type [T-5](#)
- TIME literal [D-11](#)
- TIME values
 - in host variables [H-6](#)
 - LOAD command [L-38](#)
- TIMEOUT option, CONTROL TABLE directive [C-84](#)
- TIMESTAMP
 - data type [T-5](#)
 - literal [D-11](#)
- TIMESTAMP values
 - in host variables [H-6](#)
 - LOAD command [L-38](#)
- Timestamps, Julian [A-60](#), [C-62](#), [J-4](#)
- TIME_FORMAT layout option [R-10](#), [T-4](#)
- Titles
 - break [B-10](#)
 - page [P-4](#)
 - report [R-6](#)
- TMF audit trails [W-5](#), [W-6](#)
- TMF transaction
 - aborting [R-22](#)
 - and COMMIT WORK [C-56](#)
 - and SECURE [S-9](#)
 - audit fix-up phase [W-6](#)
 - AUTOWORK [T-8](#)
 - BEGIN WORK statement [B-3](#)
 - beginning [B-3](#)
 - TMF transaction (continued)
 - commit phase [W-7](#)
 - committing [C-56](#)
 - CONTINUE statement not allowed [C-71](#)
 - control statements [T-6](#)
 - copying data from [C-124](#)
 - description of [T-6](#)
 - displaying status [E-3](#)
 - ending [C-56](#)
 - file recovery protection [A-44](#)
 - IDs [T-11](#)
 - initialization and load phase [W-6](#)
 - limit per table [L-13](#)
 - processing concepts [T-6](#)
- TMF transaction utility
 - AUTOWORK [T-8](#)
 - LOAD [L-33](#)
 - PURGE [P-33](#)
 - PURGEDATA [P-36](#)
 - SECURE [S-8](#)
 - VERIFY [V-4](#)
- TO [R-3](#)
- TOTAL command [T-9](#)
- Transaction ID, displaying current [E-3](#)
- Transaction integrity, audited tables [A-76](#)
- Transaction Management Facility
 - See TMF transaction
- TRANSIDS catalog table [T-11](#)
- TRANSID, in TRANSIDS table [T-11](#)
- TRIM function [T-11](#)
- TRIM option, LOAD command [L-26](#)
- TRUNC option, LOAD command [L-31](#)
- Truncating display line [S-42](#)
- Two-way partition split, ALTER TABLE statement [A-41](#)
- TYPE AS clause, DATETIME HOST VARIABLE [H-6](#)
- Type of object, displaying [F-15](#)

U

UNAVAILABLE PARTITION option,
CONTROL TABLE directive [C-83](#)

UNDERLINE_CHAR layout option [R-10](#),
[U-1](#)

UNION columns [S-25](#)

UNION operator

and GROUP BY clause [S-27](#)

and HAVING clause [S-27](#)

and ORDER BY clause [S-26](#)

and shorthand views [C-170](#)

SELECT statement [S-23](#)

UNION ALL and associativity [S-27](#)

using [S-25](#)

UNIQUE clause, CREATE INDEX
statement [C-143](#)

UNIQUE index

and duplicate values [C-143](#)

and null values [C-143](#)

description of [C-143](#)

keys for [I-9](#)

UNIQUEENTRYCOUNT in COLUMNS
table [C-46](#)

UNIQUEVALUE, in INDEXES table [I-10](#)

UNITS specifier [E-22](#)

UNLOADIN option, LOAD command [L-27](#)

UNLOADOUT option, COPY
command [C-121](#)

UNLOCK TABLE statement

description of [U-1](#)

lock release summary [L-48](#)

lock release summary [L-48](#)

Unnamed parameter [P-13](#)

Unpartitioned table [P-19](#)

Unstructured files [F-8](#)

UNSTRUCTURED option

COPY command [C-117](#)

LOAD command [L-22](#)

Unsupported data types [L-36](#)

unsupported Data types

COMPLEX [L-36](#)

LOGICAL [L-36](#)

UPDATE operations

buffered [C-82](#)

file organization dependent [F-8](#)

UPDATE statement

and scale [S-46](#)

and SETSCALE [S-46](#)

column privileges [S-68](#)

description of [U-3](#)

lock release summary [L-48](#)

table privileges [S-68](#)

UPDATE STATISTICS statement

concurrent DML operations [C-65](#)

description of [U-7](#)

Updateable primary keys [S-69](#)

UPGRADE CATALOG command [U-11](#)

UPGRADE SYSTEM CATALOG
command [U-13](#)

UPSHIFT clause, and data type [D-3](#)

UPSHIFT function [U-14](#)

UPSHIFT in COLUMNS table [C-47](#)

UPSHIFT option

COPY command [C-117](#)

LOAD command [L-23](#)

USAGES catalog table

and DISPLAY USE OF [D-54](#), [D-55](#)

and program invalidation [P-29](#)

description of [U-15](#)

Use of object, displaying [D-54](#)

USEDATALOGNAME, in USAGES
table [U-15](#)

USED OBJNAME, in USAGES table [U-15](#)

USED OBJTYPE, in USAGES table [U-15](#)

User groups [S-11](#), [S-12](#)

User IDs [S-12](#)

User number [S-12](#)

USER option, FILEINFO command [F-9](#)

USERIDin PROGRAMS table [P-31](#)in TABLES table [T-2](#)**User-defined primary key**description of [C-158](#), [U-16](#)length limit [L-13](#)specifying [C-158](#)**USESQLNULLS option**COPY command [C-117](#)LOAD command [L-23](#)**USING clause**CURRENT_TIMESTAMP function [E-8](#)EXECUTE statement [E-7](#)**USING DESCRIPTOR clause**EXECUTE statement [E-7](#)FETCH statement [F-3](#)OPEN statement [O-5](#)**USING host variables clause** [O-5](#)**USINGCATALOGNAME**, in USAGES table [U-15](#)**USINGOBJNAME**, in USAGES table [U-15](#)**USINGOBJTYPE**, in USAGES table [U-15](#)**Utilities**APPEND [A-52](#)APPENDCANCEL [A-56](#)APPENDRESTART [A-58](#)CLEANUP [C-21](#)CONVERT [C-94](#)COPY [C-113](#)description of [U-16](#)DISPLAY USE OF [D-54](#)DUP [D-69](#)FILEINFO [F-9](#)FILENAMES [F-25](#)FILES [F-26](#)FUP [F-33](#)LOAD [L-18](#)PURGE [P-33](#)PURGEDATA [P-36](#)SECURE [S-7](#)**Utilities (continued)**VERIFY [V-2](#)Utility operations, and concurrency [C-68](#)**V**Valid plan [P-22](#)Validation, program file [P-28](#)**VALIDDATA**in BASETABS table [B-1](#)in INDEXES table [I-10](#)**VALIDDEF**in BASETABS table [B-1](#)in INDEXES table [I-10](#)in VIEWS table [V-10](#)VALID, in PROGRAMS table [P-31](#)Value substitution [P-11](#)**Values**compatible [I-16](#)matching [L-2](#)search conditions for [S-5](#)searching for existing [E-12](#)selecting a range of [B-4](#)selecting from a list [I-1](#)VARCHARS option, CONVERT command [C-97](#)VARCHAR_WIDTH layout option [R-10](#), [V-1](#)**Variables**in host programs [H-6](#)SQLCODE [S-67](#)Variable-length string, and CONVERT command [C-97](#)VARIN option, LOAD command [L-27](#)VAROUT option, COPY command [C-121](#)**VERIFIEDWRITES**file attribute [V-1](#)in FILES table [F-27](#)**VERIFY command**authorization requirement [V-3](#)description of [V-2](#)

VERSION

in CATALOGS table [C-11](#)

in VERSIONS table [V-9](#)

Version

description of [V-6](#)

displaying [E-3](#)

information from FILEINFO [F-16](#)

information in catalogs [V-9](#)

numbers [V-6](#)

Version management

Data Status Language [D-67](#)

DOWNGRADE CATALOG [D-58](#)

DOWNGRADE SYSTEM
CATALOG [D-61](#)

GET CATALOG OF SYSTEM [G-1](#)

GET VERSION [G-2](#)

GET VERSION OF PROGRAM [G-4](#)

UPGRADE CATALOG [U-11](#)

UPGRADE SYSTEM CATALOG [U-13](#)

VERSION option, FILEINFO
command [F-16](#)

VERSIONS catalog table [V-9](#)

VERSIONUPGRADETIME

in CATALOGS table [C-11](#)

in VERSIONS table [V-9](#)

Vertical printing of detail line [D-51](#)

vi text editor [F-8](#)

VIEW option, DUP command [D-74](#)

VIEWNAME, in VIEWS table [V-10](#)

Views

ALTER VIEW statement [A-50](#)

altering security attributes of [A-50](#)

AUDIT file attribute [A-74](#)

catalog description of [T-2](#), [V-10](#)

column limit [C-170](#), [L-6](#)

CREATE VIEW statement [C-166](#)

definition text limit [L-13](#)

description [V-9](#)

displaying physical characteristics [F-9](#)

DROP statement [D-65](#)

Views (continued)

dropping [P-34](#), [P-35](#)

duplicating [D-69](#), [D-75](#)

grouped [S-19](#), [S-24](#)

inserting with check option [I-17](#)

limit per table [L-13](#)

locking [L-45](#)

names [G-7](#), [V-9](#)

ownership changes [A-50](#)

protection [L-14](#), [P-32](#)

purging [D-64](#)

renaming [A-50](#)

row definition [I-25](#)

security changes [A-50](#)

selecting data from [S-19](#), [S-24](#)

shorthand [C-170](#), [S-48](#)

similarity [S-55](#)

specifying security attributes [C-168](#)

table security dependencies [A-38](#)

unlocking nonaudited [U-1](#)

versions [V-7](#)

VIEWS catalog table [V-10](#)

VIEWTEXT, in VIEWS table [V-10](#)

Virtual sequential block buffering (VSBB)

cursor operation [C-91](#)

description of [C-68](#)

effect on concurrency [C-68](#)

in EXPLAIN report [E-19](#)

syntax [C-82](#)

using [C-89](#)

VLM, parameter for FastSort [Z-4](#)

Volume

current default [Z-3](#)

displaying current default [E-3](#)

setting default [V-11](#)

VOLUME command [V-11](#)

VSBB

See Virtual sequential block buffering

W

WAIT IF LOCKED option, CONTROL TABLE directive [C-82](#)

Wait time for lock requests [C-84](#)

Warnings

1618 [C-53](#), [C-56](#), [C-72](#), [W-7](#)

1619 [C-53](#), [C-56](#), [C-70](#), [C-71](#), [C-72](#), [W-7](#)

8239 [C-83](#)

displaying [S-42](#)

WARNINGS option, SET SESSION command [S-42](#)

WHENEVER directive [W-1](#)

WHERE clause

DELETE statement [D-39](#)

description of [W-2](#)

UPDATE statement [U-3](#)

WHERE CURRENT OF clause

DELETE statement [D-39](#)

UPDATE statement [U-4](#)

Widows, preventing in report [D-49](#)

Width of report [L-53](#)

Wild-card characters

in LIKE predicate [L-3](#)

in qualified filesets [Q-2](#)

WINDOW layout option [R-10](#), [W-2](#)

WITH CHECK OPTION clause, CREATE VIEW statement [C-169](#)

WITH DATA MOVEMENT clause

ALTER INDEX [A-18](#)

ALTER TABLE [A-37](#)

WITH HEADINGS clause, CREATE VIEW statement [C-169](#)

WITH HELP TEXT clause, CREATE VIEW statement [C-169](#)

WITH SHARED ACCESS clause, ALTER INDEX statement [A-17](#)

WITH SHARED ACCESS clause, CREATE INDEX statement [C-146](#)

WITH SHARED ACCESS option [W-4](#)

WITHCHECKOPTION, in VIEWS table [V-10](#)

Working attribute set [D-34](#)

WRAP option, SET SESSION command [S-42](#)

Wrapping display line [S-42](#)

Write access [S-11](#), [S-14](#)

Z

ZYQ names [O-6](#)

Special Characters

! (exclamation point) command [Z-1](#)

+> prompt [S-60](#)

.. prompt [S-60](#)

= [Z-18](#)

=_AUDSERV_XSWAP_node DEFINE [Z-2](#)

=_DEFAULTS_DEFINE [Z-3](#)

=_SORT_DEFAULTS_DEFINE [Z-4](#)

=_SQL_AUD_node define [Z-14](#)

=_SQL_CAT_HEAP_LIMIT_DEFINE [Z-6](#)

=_SQL_CAT_node define [Z-14](#)

=_SQL_CI2_node define [Z-14](#)

=_SQL_CMP_CPUS_node DEFINE [Z-7](#)

=_SQL_CMP_DOUBLE_SBB_OFF
DEFINE [Z-8](#)

=_SQL_CMP_DOUBLE_SBB_ON
DEFINE [Z-9](#)

=_SQL_CMP_EQ_LIMIT_DEFINE [Z-10](#)

=_SQL_CMP_EVENT_DEFINE [Z-11](#)

=_SQL_CMP_EVENT_NO0_DEFINE [Z-12](#)

=_SQL_CMP_node define [Z-14](#)

=_SQL_CMP_NO_KS_MJOIN
DEFINE [Z-14](#)

=_SQL_EXE_DOUBLE_SHUTOFF
DEFINE [Z-15](#)

=_SQL_EXE_ESPS_CK_CMOM
DEFINE [Z-16](#)

=_SQL_EXE_USE_SWAPVOL
DEFINE [Z-17](#)

=_SQL_MSG_node DEFINE [Z-18](#)

=_SQL_RECGEN_node DEFINE [Z-20](#)

=_SQL_TM_node_vol DEFINE [Z-20](#)

=_SQL_UTL_node define [Z-14](#)

>> prompt [S-60](#)

? unnamed parameter [P-13](#)

