StorHouse®

StorHouse SQL Reference Manual

StorHouse/RM Release 3.2

Publication Number
900111 Rev. I

March 21, 2002

FileTek

All rights reserved. No part of this publication may be reproduced, translated, stored in any electronic retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of FileTek, Inc.

This publication Copyright © 1996-2002 by FileTek, Inc., Rockville, MD
Publication Number: 900111 Rev. I

NOTE: U.S. GOVERNMENT USERS

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or the Commercial Computer Software - Restricted Rights clause at 48 CFR 52.227-19, as applicable. Unpublished-rights reserved under the copyright laws of the United States. The contractor/manufacturer is:

FileTek, Inc.
9400 Key West Avenue
Rockville, Maryland 20850

Information in this document is subject to change without notice and does not represent a commitment on the part of FileTek, Inc. Further, FileTek, Inc. reserves the right to supplement the document with information not available at the time of creation of the document. FILETEK, INC. PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, AND CANNOT WARRANT THE RESULTS YOU MAY OBTAIN USING THE DOCUMENT. IN NO EVENT SHALL FILETEK, INC. BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OR DATA, INTERRUPTION OF BUSINESS, OR FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND, EVEN IF FILETEK, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES ARISING FROM ANY DEFECT OR ERROR IN THIS PUBLICATION. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

FileTek and StorHouse are registered U.S. trademarks of FileTek, Inc. VRAM is a U.S. trademark of FileTek, Inc. All other brand or product names are trademarks or registered trademarks of their respective owners.

Documentation for FileTek’s StorHouse product. Protected by the following U.S. Patents: 4,864,572; 5,247,660; 5,727,197; 6,049,804. Other patents pending.

FileTek

Contents

Welcome xix

StorHouse family of products xix
StorHouse/SM xix
StorHouse/RM xx
Control Center xx

Purpose of this document xx

Intended audience xxi

Contents xxi

Related documentation xxii

Conventions xxiii

Chapter 1: Introduction 1-1

About StorHouse SQL 1-1

Ways to submit StorHouse SQL 1-2

Types of StorHouse SQL 1-3
Static SQL 1-3
Dynamic SQL 1-4

SQL standard differences 1-4
Through a FileTek interface 1-5
In an application program 1-5
Through a gateway 1-5

Through a federator 1-6

Types of queries 1-6
Selection 1-6
Extraction 1-7
Join 1-7
Join types 1-7
Inner-join 1-7
Left outer-join 1-7
Equi-join 1-8
Self-join 1-8
Cartesian product 1-8
Join methods 1-8
Nested loop 1-8
Hybrid IN 1-9
Subquery 1-9

StorHouse SQL conventions 1-10
SQL format conventions 1-10
SQL example conventions 1-11
Spaces in StorHouse SQL 1-12
Comments in StorHouse SQL 1-12
SQL-style comment 1-13
C-style comment 1-13
C++-style comments 1-13

Chapter 2: Elements of StorHouse SQL 2-1

Keywords 2-1

Clauses 2-2

Predicates 2-2

Functions 2-2

Expressions 2-3

User-supplied names 2-4

Account IDs 2-4
Alias names 2-5
Cursor names 2-5
Database names 2-6
Database user component names 2-6
Host variable names 2-7
LOB locator variables 2-7
LOB file reference variables 2-8
Substitution markers 2-8

Delimited SQL identifiers 2-8
Delimited account IDs 2-9
Qualified and delimited user table names 2-10
Delimited volume set and file set names 2-10
Delimited user tablespace names 2-10

Literals 2-11
Character literals 2-11
Date literals 2-12
Time literals 2-12
Timestamp literals 2-13
Integer literals 2-13
Floating-point literals 2-13
Decimal literals 2-14
Hexadecimal literals 2-14

Format strings 2-14
Date format strings 2-15
Time format strings 2-16

Operators 2-17
Set operators (UNION and UNION ALL) 2-18
Logical operators (AND, OR, NOT) 2-20
Arithmetic operators (+, -, *, /, unary) 2-21
Comparison operators (=, <>, <, >, <=, >=) 2-23
Concatenation operator (||) 2-23

NULL values 2-24

Special registers 2-25

USER special register 2-25
SYSDATE special register 2-25
SYSTIME special register 2-26

Chapter 3: StorHouse data types 3-1

About StorHouse data types 3-1

Database data types 3-2
Descriptions of database data types 3-3
Database data types and functions 3-14
StorHouse and local database data types 3-16

Loader and unloader data types 3-17
List of loader and unloader data types 3-18
Date and time formats of input and result data 3-19

Host language data types 3-21

Data type conversion 3-23
Conversion of loader data types 3-23
Conversion of unloader data types 3-26
Conversion of literals 3-28
Conversion of function data types 3-29

Chapter 4: StorHouse SQL statements 4-1

About StorHouse SQL statements 4-1

List of StorHouse SQL statements 4-2

Categories of SQL 4-5

ALTER TABLE SPACE 4-7
Format 4-7
Examples 4-10

BEGIN and END DECLARE SECTION 4-11
Format 4-11

Declaring host and indicator variables 4-12
ESQL format 4-13
C language format 4-13
Examples 4-14
Declaring arrays 4-14
ESQL format to declare an array 4-15
C language format to declare a non-char array 4-15
C language format to declare a char array 4-15
Examples 4-16
Declaring type definitions for variables and arrays 4-16
ESQL format to declare a new data type for a variable 4-16
ESQL format to declare a new data type for an array 4-17
Examples 4-18

CLOSE 4-19
Format 4-19
Example 4-19

COMMIT WORK 4-20
Format 4-20
Examples 4-21

CONNECT 4-22
Format 4-22
Example 4-23

CREATE INDEX 4-24
Format 4-24
Examples 4-26

CREATE SYNONYM 4-27
Format 4-27
Example 4-28

CREATE TABLE 4-29
Format 4-29
Examples 4-33

CREATE TABLE SPACE 4-35
Format 4-35

Examples 4-39

CREATE VIEW 4-40
Format 4-41
Example 4-41

DECLARE 4-42
Format 4-42
Examples 4-43

DELETE 4-44
Format 4-44
Example 4-45

DESCRIBE 4-46
DESCRIBE BIND VARIABLES 4-46
Format 4-46
Example 4-47
DESCRIBE SELECT LIST 4-47
Format 4-48
Example 4-48

DISCONNECT 4-49
Format 4-49
Examples 4-49

DROP INDEX 4-50
Format 4-50
Example 4-51

DROP SYNONYM 4-52
Format 4-52
Example 4-52

DROP TABLE 4-53
Format 4-53
Example 4-53

DROP TABLE SPACE 4-54
Format 4-54
Example 4-55

DROP VIEW 4-56
Format 4-56
Example 4-56

EXECUTE 4-57
Format 4-58
Examples 4-59

EXECUTE IMMEDIATE 4-60
Format 4-60
Example 4-61

FETCH 4-62
Format 4-63
Examples 4-64

FREE LOCATOR 4-66
Format 4-66
Example 4-66

GRANT 4-67
Format 4-67
Examples 4-69

INSERT 4-70
Format 4-70
Example 4-71

OPEN 4-72
Format 4-73
Examples 4-74

PREPARE 4-75
Format 4-75
Examples 4-76

REVOKE 4-77
Format 4-77
Examples 4-79

ROLLBACK WORK 4-80
Format 4-81

Example 4-81

SELECT 4-82
Format 4-82
INTO clause 4-85
Format 4-85
Examples 4-85
FROM clause 4-86
Format 4-86
Example 4-87
WHERE clause 4-88
Format 4-88
Example 4-88
GROUP BY clause 4-89
Format 4-89
Example 4-89
HAVING clause 4-90
Format 4-90
Example 4-90
ORDER BY clause 4-91
Format 4-91
Examples 4-92
FOR clause 4-93
Format 4-93
Joins 4-94
Sample tables 4-95
Performing multiple inner-join operations 4-96
Performing a left outer-join 4-97
Changing the table order 4-98
Using parentheses to specify join order 4-99
Combining inner-join and outer-join operations 4-100
Using a WHERE clause with a join 4-101
Explanation of query 1 4-101
Explanation of query 2 4-103

SET CONNECTION 4-105
Format 4-105

Example 4-105

UPDATE 4-106
Format 4-106
Example 4-107

VALUES INTO 4-108
Format 4-108
Example 4-108

WHENEVER 4-109
Format 4-109
Example 4-110

Chapter 5: StorHouse SQL predicates 5-1

About StorHouse SQL predicates 5-1
Predicate order 5-2
Comparisons of CHAR and VARCHAR fields with blanks 5-2

Basic predicate 5-3
Format 5-3
Example 5-4

Complex predicate 5-5
Examples 5-6

Quantified predicate 5-7
Format 5-7
Example 5-8

BETWEEN 5-9
Format 5-9
Example 5-9

EXISTS 5-10
Format 5-10
Example 5-10

IN 5-11
Format 5-11

Example 5-11

LIKE 5-12
Format 5-12
Examples 5-13

NULL 5-14
Format 5-14
Examples 5-14

Chapter 6: StorHouse SQL functions 6-1

About StorHouse SQL functions 6-1
Aggregate functions 6-1
Scalar functions 6-2

ABS 6-5
Format 6-5
Example 6-6

ADD_MONTHS 6-7
Format 6-7
Example 6-7

ASCII 6-8
Format 6-8
Example 6-8

AVG 6-9
Format 6-9
Example 6-10

BIT_LENGTH 6-11
Format 6-11
Example 6-11

BLOB 6-12
Format 6-12
Example 6-12

CHAR_LENGTH 6-13

Format 6-13
Example 6-13

CHR 6-14
Format 6-14
Example 6-14

CLOB 6-15
Format 6-15
Example 6-15

CONCAT 6-16
Format 6-16
Examples 6-17

COUNT 6-18
Format 6-18
Example 6-18

DAYOFMONTH 6-19
Format 6-19
Example 6-19

DAYOFWEEK 6-20
Format 6-20
Example 6-20

DAYOFYEAR 6-21
Format 6-21
Example 6-21

DAYS 6-22
Format 6-22
Example 6-22

DECODE 6-23
Format 6-23
Example 6-24

GREATEST 6-25
Format 6-25
Example 6-25

HOUR 6-26
Format 6-26
Example 6-26

INITCAP 6-27
Format 6-27
Example 6-27

INSTR 6-28
Format 6-28
Example 6-29

LAST_DAY 6-30
Format 6-30
Example 6-30

LEAST 6-31
Format 6-31
Example 6-31

LENGTH 6-32
Format 6-32
Example 6-32

LOWER 6-33
Format 6-33
Example 6-33

LPAD 6-34
Format 6-34
Examples 6-35

LTRIM 6-36
Format 6-36
Example 6-36

MAX 6-37
Format 6-37
Example 6-37

MIN 6-38
Format 6-38

Example 6-38

MINUTE 6-39
Format 6-39
Example 6-39

MONTH 6-40
Format 6-40
Example 6-40

MONTHS_BETWEEN 6-41
Format 6-41
Example 6-41

NEXT_DAY 6-42
Format 6-42
Example 6-42

NVL 6-43
Format 6-43
Example 6-43

OCTET_LENGTH 6-44
Format 6-44
Example 6-44

OVERLAY 6-45
Format 6-45
Example 6-46

POSITION 6-47
Format 6-47
Example 6-47

QUARTER 6-48
Format 6-48
Example 6-48

RPAD 6-49
Format 6-49
Example 6-50

Contents

RTRIM 6-51
Format 6-51
Example 6-51

SECOND 6-52
Format 6-52
Example 6-52

SUBSTR 6-53
Format 6-53
Example 6-54

SUM 6-55
Format 6-55
Example 6-56

TO_CHAR 6-57
Format 6-57
Example 6-57

TO_DATE 6-58
Format 6-58
Example 6-58

TO_HEX 6-59
Format 6-59
Examples 6-60

TO_NUMBER 6-61
Format 6-61
Example 6-61

TO_TIME 6-62
Format 6-62
Example 6-62

TRANSLATE 6-63
Format 6-63
Example 6-64

TRIM 6-65
Format 6-65

Example 6-66

UPPER 6-67
Format 6-67
Example 6-67

WEEK 6-68
Format 6-68
Example 6-68

YEAR 6-69
Format 6-69
Example 6-69

Appendix A: SQL status codes A-1

Appendix B: StorHouse SQL reserved words B-1

Appendix C: Deprecated syntax C-1

ALTER TABLE SPACE C-2
Format C-2
Example C-5

CREATE TABLE SPACE C-6
Format C-6
Examples C-9

Joins C-11
Format C-12
Examples C-13

Index

Contents

FileTek

Welcome

StorHouse® Structured Query Language (SQL) is a subset of the American National Standards Institute (ANSI) SQL. StorHouse SQL includes extensions that are specific to defining and accessing database data on the FileTek® StorHouse system.

StorHouse family of products

StorHouse is the FileTek enterprise-wide solution for managing the capture, storage, movement, and access of gigabytes to petabytes of relational and non-relational detail data. StorHouse technology combines industry-leading, scalable storage devices and Open System processors with specialized storage management and relational database management system (RDBMS) software components.

StorHouse/SM

StorHouse/SM, the storage management component, controls a hierarchy of storage devices comprised of cache, redundant array of independent disk (RAID), erasable and write-once-read-many (WORM) optical disk jukeboxes, and automated tape libraries. StorHouse/SM is also responsible for automating critical system management tasks, like data migration, backup, and recovery. StorHouse/SM comes standard with all StorHouse systems.

StorHouse/RM

StorHouse/RM, the RDBMS component, works in conjunction with StorHouse/SM to specifically administer the storage, access, and movement of relational data. StorHouse/RM provides row-level SQL access to high volumes of detail data on any layer—including tape—in the StorHouse storage hierarchy. SQL access is available from different platforms through a variety of industry-standard protocols. StorHouse/RM runs on Sun™ Solaris™ and Hewlett-Packard HP-UX platforms.

Control Center

StorHouse *Control Center* (CC) is the FileTek Windows®-based network computing system for providing administrative control of the StorHouse family of products. Control Center works with StorHouse/SM release 4.2 and higher and consists of one or more Control Center servers that communicate with Control Center clients over a TCP/IP network. The *Control Center server*, which runs on Windows NT, XP Pro, and 2000 platforms, provides network connectivity to StorHouse. The *Control Center clients*, which run on Windows 95, 98, 2000, XP Pro, and NT platforms, consist of one or more graphical user interface (GUI) modules for performing StorHouse system and database administration tasks, configuring and managing Control Center servers, and analyzing and monitoring StorHouse activity and performance.

Purpose of this document

The *StorHouse SQL Reference Manual* serves as a reference for StorHouse SQL. This manual contains SQL formats and examples and provides guidelines for using StorHouse SQL.

Intended audience

The *StorHouse SQL Reference Manual* is intended for users, application programmers, and system and database administrators. This manual assumes that you understand SQL and StorHouse/RM concepts.

Contents

This publication contains the following chapters and appendixes:

- Chapter 1, “Introduction,” provides basic information about StorHouse SQL such as ways to invoke SQL, types of SQL, and conventions used to illustrate StorHouse SQL.
- Chapter 2, “Elements of StorHouse SQL,” describes the components that are common to many SQL statements.
- Chapter 3, “StorHouse data types,” describes different categories of data types and provides data type conversion information.
- Chapter 4, “StorHouse SQL statements,” contains descriptions, formats, and examples of StorHouse SQL statements.
- Chapter 5, “StorHouse SQL predicates,” contains descriptions, formats, and examples of StorHouse SQL predicates.
- Chapter 6, “StorHouse SQL functions,” contains descriptions, formats, and examples of StorHouse SQL aggregate and scalar functions.
- Appendix A, “SQL status codes,” lists the StorHouse SQL status codes and associated text you receive after submitting StorHouse SQL statements.
- Appendix B, “StorHouse SQL reserved words,” lists the StorHouse SQL reserved words that you cannot use to name various SQL elements.

- Appendix C, “Deprecated syntax,” contains the deprecated format of ALTER TABLE SPACE, CREATE TABLE SPACE, and joins.

Related documentation

It may be helpful to be familiar with the material in these documents:

- The *StorHouse SQL Quick Reference*, publication number 900122, provides a summary of the material in the *StorHouse SQL Reference Manual*.
- The *StorHouse Database Administration Guide*, publication number 900108, describes StorHouse database concepts and explains how to create user tables and indexes, manage accounts and privileges, set up user tablespaces, and perform other StorHouse system and database administration tasks.
- The *StorHouse ESQL Manual*, publication number 900121, explains how to use StorHouse SQL in application programs.
- The *FileTek MVS Data Loader Utility Manual*, publication number 900109, describes how to load data into StorHouse user tables from an MVS environment.
- The *FileTek FTP Data Loader Manual*, publication number 900115, explains how to load data into StorHouse user tables from UNIX®, VAX, or other hosts using your standard File Transfer Protocol (FTP) client software.
- The *FileTek FTP Data Unloader Manual*, publication number 900137, explains how to unload data from StorHouse databases using FTP. It describes the UNLOAD control statement you prepare to format result data, the SELECT statement you prepare to select the data to unload, and the subset of FTP commands you use to transfer control information and to receive result data.

- The *StorHouse/RM Glossary*, publication number 900112, defines the terminology used in the StorHouse/RM User Document Set.
- The StorHouse *Messages and Codes Manual*, publication number 900032, lists all StorHouse system and host software messages.
- The *StorHouse/UDB Link Installation and Configuration Manual*, publication number 900163, explains how to install the StorHouse/UDB Link and how to configure IBM® DB2® Universal Database (DB2 UDB) to work with StorHouse. It also discusses security management, data modeling, and DB2 components that support federation.
- The *Microsoft SQL Server 7.0-to-StorHouse Connectivity Guide*, publication number 900158, describes how to access StorHouse/RM from a Microsoft SQL Server application using the MERANT™ SequeLink® Server software. This document provides instructions on configuring linked servers with StorHouse/RM so that Microsoft SQL customers can develop applications that use StorHouse/RM as a transparent data source.

Conventions

This book uses the following notational conventions:

Convention	Meaning
Helvetica font	StorHouse SQL formats and examples
Italics	New terms, emphasized text, and publication titles

See “StorHouse SQL conventions” on page 1-10 for specific StorHouse SQL conventions.

Chapter 1

Introduction

This chapter presents an overview of StorHouse SQL. It describes:

- Basic uses of StorHouse SQL
- Ways to invoke StorHouse SQL
- Static and dynamic SQL
- SQL standard differences
- Types of queries
- Syntax rules for StorHouse SQL formats and examples

About StorHouse SQL

Structured Query Language (SQL) is a standardized language for defining and manipulating data in relational database management systems. *StorHouse SQL* is the version of SQL that is native to StorHouse. It consists of a subset of American National Standards Institute (ANSI) SQL plus extensions defined by FileTek to support additional capabilities.

You use StorHouse SQL to access database data on StorHouse. This information makes up the active archive portion of your local database or data warehouse. Typically, you do not modify this data. Instead, you retain this data over time to satisfy legal commitments or to grow your corporate data warehouse.

You also use StorHouse SQL to perform StorHouse database administration tasks. For example, you can:

- Create StorHouse database user components like user tables, user tablespaces, indexes, views, and synonyms.
- Control access to database information by granting and revoking privileges for StorHouse accounts.
- Query system tables to obtain information about database components.

Ways to submit StorHouse SQL

You can submit StorHouse SQL statements several ways:

- Embed them in C or C++ application programs. This SQL is called *static SQL*, described on page 1-3. Follow the guidelines in the *StorHouse ESQL Manual*.
- Prepare and execute them dynamically with C and C++ application programs. This SQL is called *dynamic SQL*, described on page 1-4. Refer to the *StorHouse ESQL Manual* for more information.
- Issue them interactively with an Interactive SQL (ISQL) tool provided by your local database. For instance, if your local database is DB2®, then you can use SQL Processor Using File Input (SPUFI) to submit StorHouse SQL.
- Issue them interactively with the ISQL tool provided by the Control Center StorHouse/Admin module. Refer to the *StorHouse/Admin Database Administrator's Quick Reference* for more information about using this FileTek ISQL tool.
- Issue them with local database applications through a database gateway such as the IBM Distributed Relational Database Architecture (DRDA) gateway for DB2 and the Open Database Connectivity (ODBC) gateway interface for Oracle®, Microsoft SQL Server 7.0, and other ODBC-enabled systems.

- Submit them through the StorHouse/UDB Link software, which allows users of IBM DB2 Universal Database (UDB) software (version 7.1 or later) to access StorHouse databases managed by StorHouse/RM release 3.0 and higher. Refer to the *StorHouse/UDB Link Installation and Configuration Manual* for more information about submitting SQL through a federator.

- Use the FileTek MVS Data Loader utility (for MVS hosts) or the FileTek FTP Data Loader (for any FTP-capable host). Refer to the *FileTek MVS Data Loader Utility Manual* and the *FileTek FTP Data Loader Manual* for more information about submitting StorHouse SQL with a FileTek data loader.

- Prepare a SELECT statement, as part of the UNLOAD statement, to select the StorHouse data that you want to copy from a StorHouse user table to a file on a client system. Refer to the *FileTek FTP Data Unloader Manual* for more information about the UNLOAD and SELECT statements.

Types of StorHouse SQL

There are two types of SQL statements: static and dynamic.

Static SQL

Static SQL (or *embedded SQL*) *statements* are embedded in an application program. You use static SQL when you know the contents of your SQL statements at program compile time. That is, you know which statements you're going to issue and the names of the tables and columns you plan to select. The only items that may change from one execution to the next are values in your search conditions.

StorHouse provides an *Embedded SQL Interface* (*ESQL*) that supports coding SQL statements in C and C++ programs. You can embed all StorHouse SQL statements in a program, placing them wherever a host language statement is

allowed. The SQL statement must begin with the prefix EXEC SQL and end with a semicolon.

Before you compile a program containing static SQL statements, those statements must be processed by the StorHouse ESQL precompiler. The *ESQL precompiler* translates your SQL statements into host language statements that include standard ESQL library calls, which are then used to build the program executable. Refer to the *StorHouse ESQL Manual* for more information about embedding SQL statements in your programs and running the StorHouse ESQL precompiler.

Dynamic SQL

Dynamic SQL statements are prepared and executed by a program at runtime. You can use dynamic SQL statements when you don't know which SQL statements you intend to execute or which columns and tables you plan to select until a program executes.

Dynamic SQL statements are not embedded in a program. Your program could, for example, build an SQL statement from terminal input placed into a character variable. A dynamic SQL statement can change several times during a program's execution.

All StorHouse SQL statements can be static (you can embed all of them in a program), but only some are dynamic. See the table on page 4-2 for a list of StorHouse SQL statements that you can prepare and execute dynamically.

SQL standard differences

SQL formats differ somewhat from one database system to another. When multiple database products are involved in the execution of a single SQL statement, the standard to which that statement must conform depends upon how the statement is presented and the type of interfaces the statement passes

through. This section describes standards for SQL submitted through a FileTek-supplied interface, an application program, a gateway, and a federator.

Through a FileTek interface

SQL presented to StorHouse through a FileTek-supplied interface must conform to the StorHouse SQL documented in this manual. These interfaces are:

- StorHouse ESQL precompiler
- FileTek MVS Data Loader utility
- FileTek FTP Data Loader
- FileTek FTP Data Unloader
- StorHouse/Admin ISQL

In an application program

SQL embedded in an application program (that is, static SQL) must always conform to the SQL standards of the database system that supplies the embedded SQL precompiler. If StorHouse also executes the statement, (for example, a SELECT statement), then that statement must also conform to StorHouse standards. Dynamic SQL appears to the precompiler as a character string and is not processed by the precompiler. Such SQL must conform only to the standards of the server that executes it.

Through a gateway

SQL submitted through a gateway, such as the StorHouse DRDA gateway, or through an industry standard API such as ODBC must conform with the standards of the server that executes it, in this case, StorHouse. Note that these interfaces also parse SQL statements but are usually tolerant of SQL variations.

Through a federator

SQL submitted through a federator, such as the IBM Datajoiner, DB2 UDB release 7.1 or later, or OracleHS must conform to the standards of the federator. Federators, then, may restructure the SQL to conform to the standards of the target database system. The best approach is to use SQL common to all database systems and to avoid inconsistently defined objects such as date-time values expressed as string literals.

Many federators have a "pass-through" feature that submits SQL to a target database system without any processing by the federator. In this case, the SQL needs to conform only to the standards of the target database. Note, however, that pass-through may inhibit optimization and limit capabilities like cross-database joins.

Types of queries

You retrieve data by submitting a *query* with a SELECT statement. The result of a query is data in the form of a table, called *result table*, *result set*, *active set*, or *answer set*. There are many ways to write a query. This section describes these specific types of queries: selection, extraction, join, and subquery.

Selection

A *selection* returns specified columns from one or more rows in a table. For instance, when you access all the information in a user table for a specific account number, then you are performing a selection.

Extraction

An *extraction* is a query that returns all rows for one or more columns of a user table. This type of query results in a *full table scan (*a search in all rows without the use of an index) or a *full segment select* (a result set consisting of one or more entire segments with the use of a range index). Depending on the size of the user table, an extraction may take too long to be practical. But because an extraction may be necessary, StorHouse provides an efficient method—called the StorHouse *extractor*—for processing full table scans and full segment selects. Queries and user tables must conform to a set of rules to qualify for extractor processing. Refer to the *StorHouse ESQL Manual* for more information about extracting data with the extractor.

Join

A *join* combines data in multiple tables or views or even within a table or view. When you join tables, you specify a column in each table and a join condition. The SELECT statement FROM clause identifies the tables to be joined and the join condition.

Join types

StorHouse supports inner-joins, left outer-joins, equi-joins, self-joins, and Cartesian products. Full outer-join and right outer-join operations are currently not supported.

Inner-join. An *inner-join* combines the matched rows of the tables. The unmatched rows are omitted from the result table.

Left outer-join. An *outer-join* also combines the matched rows of the tables, and for unmatched rows, preserves the values of the table to the left of the join operator, combining them with NULL values for the table to the right of the join operator.

Equi-join. An *equi-join* uses predicates that specify equalities to join a column from one table with a column from another table. For example, you can select the customer number, name, order number, and order date from the CUSTOMERS and ORDERS tables for all customer numbers contained in both tables.

Self-join. A *self- or auto-join* joins a table with itself. For example, you can select all customer names and numbers that are from the same city as a specified customer in the table.

Cartesian product. A *Cartesian product* joins tables in their entirety. The number of rows in a Cartesian product equals the number of rows in one table multiplied by the number of rows in the second table. For example, if you're joining two tables and each table contains 100 rows, then the result set is a Cartesian product with 10,000 rows (100 x 100). To avoid a Cartesian product, qualify a join with a WHERE clause.

Join methods

A *join method* is the series of steps that the optimizer performs to join tables. StorHouse/RM supports two join methods:

- Nested loop
- Hybrid IN

The optimizer chooses the most efficient join method for the query but may consider one type over the other when certain conditions are met. For instance, the optimizer may choose a hybrid IN method when the query is an equi-join and the inner table has a value index on the join column.

Nested loop. A *nested loop* method:

- Scans the outer table once for qualifying rows
- Scans the inner table as many times as the number of qualifying rows in the outer table

The optimizer may perform a nested loop when the number of qualifying rows in the outer table is small, when the predicate is not an equals-type, or when the join column of the inner table does not have an index.

Hybrid IN. A *hybrid IN* method for an inner-join works as follows:

- Scans the outer table for qualifying rows and builds a temporary result set sorted by join key
- Scans the index of the inner table to obtain the tuple IDs (TIDs) of the qualifying rows in the inner table
- Reads the inner table in TID order and joins the matching rows to those from the temporary result set of the outer table

A hybrid IN for an outer-join adds these steps to a hybrid IN inner-join:

- Flags the qualifying rows from the outer table when they match any rows from the inner table
- Scans the flags to find outer rows that were unmatched
- Projects those unmatched rows with NULL values in the inner row columns

Subquery

A *subquery* (also called *nested query* or *subselect)* is a form of the SELECT statement that appears in another SQL statement. The rows returned by the subquery are used by the higher-level SQL statement. Some of the main uses of subqueries are to define the set of rows to be included in a view, to provide values for conditions in WHERE and HAVING clauses of the SELECT statement, and to optimize higher-level queries by reducing the size of the result set, for instance, in predicates.

Note the following restrictions:

- Set operators (UNION and UNION ALL) are not allowed in subqueries.
- A join condition cannot contain a subquery.

StorHouse SQL conventions

This section describes the conventions of StorHouse SQL formats and examples and describes rules for including spaces and comments in SQL statements.

SQL format conventions

The following table can help you interpret the symbols and other conventions used in SQL formats.

Format conventions

Convention	Description
UPPERCASE	Uppercase terms indicate keywords or specific values that are part of the syntax.
lowercase	Lowercase terms refer to user-supplied values, component names, or a user-specified identifier that has specific values associated with it.
( ) , * - ; : . + ‘	These characters are part of the syntax.
{ }	Braces indicate that the item shown is required. If the format shows several options within this set of symbols, you must specify one of the options.
[ ]	Square brackets indicate that the item shown is optional.
|	A vertical bar separates alternatives. You can specify only one of the alternatives shown.
...	Ellipsis points indicate that you can repeat the part of the statement preceding them any number of times.

For example, the following SQL format (a simplified SELECT statement) uses the conventions defined in the preceding table:

```
SELECT column_name
[FROM [owner.]{table_name | view_name}]
[WHERE condition]
```

In this example:

- SELECT, FROM, and WHERE are shown in uppercase because they are StorHouse SQL keywords.
- column_name, owner, table_name, view_name, and condition are shown in lowercase because they are user-specified components.
- table_name and view_name are enclosed in braces ({ }) and separated by a vertical bar (|) because one of them is required to complete the statement. The braces indicate a requirement, while the vertical bar indicates a choice of items within the braces.
- owner, the FROM clause, and the WHERE clause are enclosed in square brackets ([ ]) because they are not required to complete this statement. They are optional.

SQL example conventions

In this manual, one or more examples follow each SQL format. For SQL statements that are both static and dynamic, the corresponding examples are shown in uppercase (even user-supplied values). For example:

```
DROP TABLE CUSTOMER
```

For SQL statements that you can embed only in a program (static only), the keywords are shown in uppercase and the user-supplied values are shown in lowercase to conform to C and C++ host language conventions. The embedded

SQL examples also contain a terminating semicolon to conform to host language conventions. For example:

```
EXEC SQL
    OPEN customer_cursor ;
```

Spaces in StorHouse SQL

A *space* is a sequence of one or more blank characters that is used as a delimiter. The syntax rules for spaces are as follows:

- User-supplied names (such as database names and table names) must not contain a space.
- Any statement or statement component can be followed by a space.
- Every numeric literal, host identifier (a name defined in a host program), or keyword must be followed by a delimiter or a space.
- If the syntax does not allow an identifier to be followed by a delimiter, that identifier must be followed by a space.

Comments in StorHouse SQL

You can include comments in StorHouse SQL. There are three kinds of comments: SQL-style, C-style, and C++-style. This section describes guidelines for including these types of comments. Any other use of comments is as indicated by the documentation for the product or application you use to submit the SQL.

SQL-style comment

You can include *SQL-style comments* in embedded SQL wherever blanks are allowed (except between EXEC SQL). These comments start with two hyphens (--) and terminate by the end of the line. SQL-style comments are not allowed within statements that are dynamically prepared (processed by PREPARE or EXECUTE IMMEDIATE).

The following example contains two SQL-style comments.

```
EXEC SQL
    SELECT names, dates --select list
    INTO :employee_name, :hire_date --output host variables
    FROM employee_data ;
```

C-style comment

You can include *C-style comments* in embedded SQL and SQL that's dynamically prepared (processed by PREPARE or EXECUTE IMMEDIATE). You can place C-style comments wherever blanks are allowed (except between EXEC SQL). These comments begin with the characters /* and end with the characters */. For example:

```
EXEC SQL
    PREPARE sel_stmt FROM 'SELECT col1, col2 /*select list*/
    FROM table1' ;
```

C++-style comments

You can include *C++-style comments* in embedded SQL wherever blanks are allowed (except between EXEC SQL). These comments begin with the characters // and terminate by the end of line. For example:

```
EXEC SQL
    SELECT names, dates //select list
    INTO :employee_name, :hire_date //output host variables
    FROM employee_data ;
```

Chapter 2

Elements of StorHouse SQL

StorHouse SQL statements contain specific elements and must conform to certain rules. This chapter defines the following elements:

- Keywords
- Clauses
- Predicates
- Functions
- Expressions
- User-supplied names
- Literals
- Format strings
- Operators
- NULL values
- Special registers

Keywords

A *keyword* is a predefined word that initiates and is reserved for specific tasks. The word SELECT, for instance, is a keyword that initiates a query. With StorHouse SQL:

- You can type keywords in uppercase, lowercase, or mixed case letters, for example, Select, SELECT, and select.

- You cannot use keywords to name database components. For example, an error occurs if you try to create a table with the name ORDER.

See Appendix B, "StorHouse SQL reserved words," for a list of keywords and other words that you cannot use for user-supplied names.

Clauses

An SQL statement is composed of one or more clauses. A *clause* begins with a keyword followed by parameters or qualifiers known as *arguments*. For instance, when creating a user tablespace with the CREATE TABLE SPACE statement, you provide one or more SUBSPACE clauses that define the storage specifications for that user tablespace. Or when you write a SELECT statement, you qualify the query by specifying an INTO, FROM, WHERE, GROUP BY, HAVING, or ORDER BY clause. The WHERE clause of the SELECT statement is also called a *restrictive clause* because it limits the number of rows returned. StorHouse SQL clauses are described in Chapter 4, "StorHouse SQL statements," with their associated SQL statements.

Predicates

A *predicate* reduces the number of rows returned by a query. You can use predicates to narrow the scope of queries so that your result sets are more precise. For example, if you are reviewing salary data and want to review a specific salary range, you can use the BETWEEN predicate to return only that salary range. The predicates supported by StorHouse SQL are defined in Chapter 5, "StorHouse SQL predicates."

Functions

A *function* is a named operation in an SQL statement, followed by one or more expressions. You use functions to derive results either from a collection of values across one or more rows of a table (using *aggregate functions* such as COUNT or

SUM) or to produce a single value from another value (using *scalar functions* such as SUBSTR). The functions supported by StorHouse SQL are defined in Chapter 6, "StorHouse SQL functions."

Expressions

An *expression* specifies a value. Expressions can consist of one or a combination of the following:

- Column names
- Operators
- Functions
- Host variables
- Special registers
- Literals

For example, the expression in the following WHERE clause includes the column name MGR_SAL, the operator >, and the integer literal 50000:

```
SELECT MGR_SAL
FROM EMPTAB
WHERE MGR_SAL > 50000
```

You can use expressions in functions and in the following:

- Condition of the WHERE and HAVING clauses of the SELECT statement
- ORDER BY clause of the SELECT statement
- VALUES clause of the INSERT statement (for SYSSMUSERS system table)
- SET clause of the UPDATE statement (for SYSSMUSERS system table)
- VALUES INTO statement
- Join condition of the SELECT statement FROM clause

A *query expression* is a SELECT statement. You can specify a query expression in the following:

- UNLOAD statement in an unload control file
- CREATE VIEW, DECLARE, and INSERT SQL statements
- Basic, quantified, EXISTS, and IN predicates

User-supplied names

For various SQL statements you assign or provide a name. For instance, when creating a user table you assign a table name in a CREATE TABLE statement. When querying that user table you provide the table name in a SELECT statement. Some of the user-supplied names include:

- Account IDs
- Alias names
- Cursor names
- Database user component names
- Database names
- Host variables

The term *SQL identifier* refers to alias names, cursor names, and database user component names.

Account IDs

An *account ID* is a name for a StorHouse account. In StorHouse SQL you provide a StorHouse account ID when:

- Specifying an owner name
- Granting or revoking a privilege
- Connecting to a StorHouse database

- Loading data
- Assigning, changing, or removing a default user tablespace for an account

An account ID consists of 1 to 12 characters that may include letters A–Z, numbers 0–9, dollar sign ($), and underscore (_). If necessary, you can enclose account IDs in double quotes (see page 2-8).

Alias names

An *alias name* is another name for a table or view. You specify an alias name when you join a table with itself (self-join). An alias name must conform to the same rules as database user component names (see page 2-6); however, it is valid only for the life of the query. You can enclose alias names in double quotes (see page 2-8).

In StorHouse SQL, an alias name typically follows the table name. The names are separated by a space. For example, in the following FROM clause, the table name is CUSTOMER and the alias names are FIRST and SECOND:

```
FROM CUSTOMER FIRST, CUSTOMER SECOND
```

Cursor names

You use *cursors* in static SQL for queries that return multiple rows. A *cursor* is a named item that points to a specific row within a set of rows and retrieves rows from that set. A *cursor name* must be unique and conform to the same rules as database user component names (see page 2-6). You can also enclose cursor names in double quotes (see page 2-8).

Database names

A StorHouse *database name* identifies a StorHouse database. A StorHouse database name is case sensitive. It must:

- Be unique
- Start with a letter
- Contain from 1 to 32 characters
- Consist of any combination of letters a–z or A–Z, numbers 0–9, and underscore (_)

If you are accessing StorHouse through your local database, then a StorHouse database name must also conform to the database naming conventions of your local database. For example, if your local database is DB2, then your StorHouse database names must be uppercase and cannot exceed 16 characters.

You cannot use case to differentiate StorHouse database names. For instance, you can't name one database CUSTOMER and a second database customer or Customer. Database names must be unique.

Database user component names

Database user components consist of user tables, columns, views, indexes, synonyms, and user tablespaces. You assign a name when you create a component and then use that name when referencing that component in an SQL statement. A database user component name:

- Cannot contain spaces
- Cannot exceed 32 characters
- Cannot be a reserved word
- Is not case sensitive unless delimited
- Starts with a letter followed by any combination of letters a–z or A–Z, numbers 0–9, and underscore (_)

You must delimit (use double quotes) database component names that do not conform to these SQL identifier conventions. See "Delimited SQL identifiers" on page 2-8 for more information about delimiting database user component names.

StorHouse/RM stores non-quoted database user component names in uppercase letters, but you can type them in any case (uppercase, lowercase, or mixed case). For instance, you can type a column called Address as ADDRESS, Address, or address.

Note: The SYS prefix is reserved for system tables. Do not use this prefix for database user component names.

Host variable names

A *host variable* is a data item declared in a host language for use within an SQL statement. A *host variable name* must follow the naming conventions of your host language. A colon (:) must precede the host variable name in StorHouse SQL, for example, :accountdba.

The following sections summarize different types of host variables. Refer to the *StorHouse ESQL Manual* for more information about declaring and using host variables.

LOB locator variables

A *LOB locator* or *locator variable* is a host variable representing a single LOB value or the result of a LOB expression. You use locator variables to identify and manipulate a LOB value at the StorHouse server or to access parts of a LOB value. A locator variable has a data type of BLOB_LOCATOR or CLOB_LOCATOR. See page 3-21 for more information about these data types.

LOB file reference variables

A *LOB file reference variable* is a host variable used to transfer a LOB value (or part of it) to a client file. A file reference variable contains the name of the client file and has a data type of BLOB_FILE or CLOB_FILE. See page 3-21 for more information about these data types.

Substitution markers

Dynamic SQL may contain *place holders*, or *substitution markers*, for host variables that are substituted in an SQL statement. A marker name doesn't have to have the same name as the host variable it represents; but it must start with a semicolon, for example, :marker_name. If you prefer, you can use the question mark symbol (?) as a place holder for a host variable instead of supplying a marker name.

Delimited SQL identifiers

A *delimited SQL identifier* is an alias name, cursor name, or database user component name enclosed in double quotes ("). You can also delimit account IDs and volume set and file set names used as a value for an SQL identifier when they don't follow SQL identifier naming conventions. FileTek recommends you follow SQL identifier naming conventions when possible. You must delimit an SQL identifier when it:

- Contains a space (non-contiguous characters)
- Consists of lowercase letters and you want to retain the case
- Begins with a character other than a letter (such as a number)
- Contains characters other than alphanumeric characters and _
- Is an SQL reserved word

StorHouse/RM converts SQL identifiers to uppercase, but it does not convert delimited SQL identifiers to uppercase. In other words, delimited SQL identifiers retain their exact case. StorHouse/RM converts account IDs as well as delimited

account IDs to uppercase. The following sections contain more information about delimited names.

Delimited account IDs

When you create StorHouse accounts, you must delimit account IDs that do not follow SQL identifier conventions. When specifying account IDs as owner names, you can type delimited account IDs in any case. For example:

```
SELECT COL1 FROM "1charlie".BILLING
```

or

```
SELECT COL1 FROM "1CHARLIE".BILLING
```

When connecting to a StorHouse database with the CONNECT statement or logging into the StorHouse FTP server, you can type delimited account IDs in any case. For example:

```
CONNECT TO 'filetek:T:remotehost.salesdb' AS 'con_2'
USER '1charlie' USING :uncle ;
```

and

```
fltksun.1> ftp alpha1 1985
Connected to alpha1.
220 alpha1 LD/FTP server ... ready.  Enter StorHouse account.
Name: 1ChaRliE
```

But when specifying account IDs as character literals in INSERT, UPDATE, and DELETE statements against metadata, you must type delimited account IDs in uppercase (because StorHouse/RM always converts accounts IDs—even delimited account IDs—to uppercase). For example:

```
INSERT INTO SYSADM.SYSSMUSERS (ACCOUNTID, DEFAULT_TS)
VALUES ('1CHARLIE','MAR1996')
```

Qualified and delimited user table names

Delimit the table name but not the owner (unless the owner name is a delimited account ID). For instance, for a user table named ORDER (which is a reserved word) owned by JAK, specify:

```
JAK."ORDER"
```

Delimited volume set and file set names

When submitting a CREATE TABLE SPACE or ALTER TABLE SPACE statement, you must delimit volume set and file set names that start with a number or _ (underscore), contain a $, or are an SQL reserved word. For example:

```
CREATE TABLE SPACE BILLINGTABLE
(SUBSPACE 1 VSET "ORDER" FSET "ORDER")
```

Delimited user tablespace names

If you delimited a tablespace name when you created a user tablespace and the name contains lowercase letters, be sure to match the case when inserting, updating, or deleting rows in the SYSSMUSERS system table. For example:

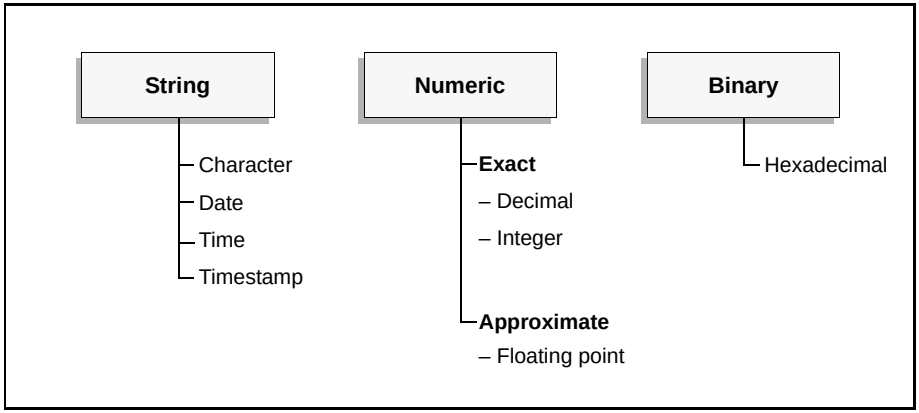
```
INSERT INTO SYSADM.SYSSMUSERS (ACCOUNTID, DEFAULT_TS)
VALUES ('USER1','Mar1996')
```

If you didn't delimit the tablespace name when you created the user tablespace, type the name in uppercase letters when inserting, updating, or deleting a row in SYSSMUSERS. For example:

```
INSERT INTO SYSADM.SYSSMUSERS (ACCOUNTID, DEFAULT_TS)
VALUES ('PUBLIC','MAR1996')
```

Literals

A *literal* (also called a *constant*) specifies a data value. You can provide literals wherever SQL syntax allows expressions. All literals have the attribute NOT NULL. Literals can be classified as string, numeric (exact and approximate), and binary.



Character literals

A *character literal* specifies any character (such as letters a–z and A–Z, numbers 0–9, spaces, and special characters). Enclose character literals in single quotes. If a character literal contains a single quote, then precede that quote with another single quote. Examples:

‘Smith’ ‘2,000’ ‘1/2/1997’ ‘VALENTINE’’S’

StorHouse/RM stores data values in the exact case in which they were loaded. You must use the same case when specifying character literals for character data values. For instance, if you loaded customer names with mixed case letters, then you must use mixed case in your SQL (for example, ‘Smith’ instead of ‘SMITH’ or ‘smith’).

Note that functions such as INITCAP and LOWER convert data formats for display and comparison purposes. The case of data, however, remains exactly as it was loaded (uppercase, lowercase, or mixed case). For example, the function INITCAP capitalizes the initial letter of a selected string for display or comparison purposes, but it does not capitalize the initial letter of your stored data.

Date literals

A *date literal* specifies a month (MM), day (DD), and year (YYYY) string in one of the formats listed in the following table. The year must be four characters. The month and day can be one or two characters. Enclose date literals in single quotes.

Date literal format	Example
M[M]/D[D]/YYYY	‘2/14/1997’
M[M]-D[D]-YYYY	’2-14-1997’
YYYY-M[M]-D[D]	‘1997-02-14’
D[D].M[M].YYYY	‘14.2.1997’

Time literals

A *time literal* specifies an hour (HH), minutes (MM), seconds (SS), and milliseconds (CCC) string in one of the formats listed in the following table. The hour, minutes, and seconds can be one digit. Milliseconds are optional. StorHouse/RM interprets milliseconds as fractional seconds up to three decimal digits of precision. Enclose time literals in single quotes.

Time literal format	Example
H[H]:M[M]:S[S].[CCC]	‘12:06:00’
H[H].M[M].S[S].[CCC]	‘12.6.0.001’

Timestamp literals

A *timestamp literal* specifies a date and time combination in one of the formats listed in the following table. You can use any date format with any time format. Separate the date from the time with a space or a dash (-). The month, day, hour, minutes, and seconds can be one or two digits. The CCCs are optional. Enclose timestamp literals in single quotes.

Date format	Time format	Example
M[M]/D[D]/YYYY	H[H]:M[M]:S[S]	‘02/07/1995 23:26:45’
YYYY-M[M]-D[D]	H[H].M[M].S[S].[CCC]	‘1995-02-07 23.26.45’
D[D].M[M].YYYY	H[H].M[M].S[S].[CCCC]	‘7.2.1995-23.26.45’
M[M]-D[D]-YYYY	H[H].M[M].S[S].[CCCCC]	‘1995-02-07-23.26.45.000010’
	H[H].M[M].S[S].[CCCCCC]	‘02/07/1995-23.26.45.00001’
		‘2-7-1995 23:26:45’

Integer literals

An *integer literal* specifies a binary integer as a signed or unsigned number with a maximum of 10 significant digits and no decimal point. Do not enclose integer literals in quotes. The range for integer literals is +2147483647 to -2147483648. Examples:

-28375 55 +999 32767

Floating-point literals

A *floating-point literal* specifies a floating-point number as two numbers separated by an E or e. This is called *E-notation*. Both numbers can include a sign, but only the first number can include a decimal point. The maximum number of digits allowed in the first number is 17. The maximum number of

digits allowed in the second number is 3. StorHouse/RM treats floating-point literals as 8-byte DOUBLE PRECISION. Examples:

15E134 2.3e5 -2.1E-2 +4.E+1

Decimal literals

A *decimal literal* specifies a decimal number as a signed or unsigned number that has a maximum of 31 digits and may include a decimal point. Examples:

+847.75 -015.20 4000. 37589477382093833.33

Hexadecimal literals

A *hexadecimal literal* represents binary data. It specifies a string of one or more hexits enclosed in single quotes and prefaced by an uppercase or lowercase X. A *hexit* is a member of the character set a–f, A–F, and 0–9. These are the valid characters for writing base-16 hexadecimal values. Examples:

x‘abc’ X‘0120ABC’

Format strings

You can specify a format string when using the TO_CHAR function to convert a given expression to character form. A *format string* consists of a *mask* that StorHouse/RM interprets and replaces with formatted date and time values. The mask is case sensitive, so if you type it in uppercase, then any character data is returned in uppercase. Enclose format strings in single quotes.

Date format strings

A date format string can contain any of the following masks along with other characters such as slashes (/) and hyphens (-). StorHouse/RM replaces the mask with values and retains any other characters.

Caution: When you specify a year mask with fewer than four digits (for instance YYY, YY, or Y), the missing digits are assumed to be zeroes. Also, TO_CHAR (a StorHouse function) assumes a uniform application of the Gregorian calendar throughout the supported date range.

Masks for date format strings

Mask	Description	Example values
CC	Century as a 2-digit number	19
YYYY	Year as a 4-digit number	1997
YYY	Last 3 digits of the year	997
YY	Last 2 digits of the year	97
Y	Last digit of the year	7
Y,YYY	Year as a 4-digit number with a comma	1,997
Q	Quarter of the year as a 1-digit number	1 (for first quarter)
MM	Month as a 2-digit number	01 (for January)
MONTH	Name of the month as a 9-character string	JANUARY, FEBRUARY
MON	First 3 characters of the month	JAN, FEB, MAR, APR
WW	Week of the year as a 2-digit number	01, 52
W	Week of the month as a 1-digit number	1, 5
DDD	Day of the year as a 3-digit number	001, 365
DD	Day of the month as 2-digit number	01, 31
D	Day of the week as 1-digit number	1 (for SUN), 7 (for SAT)
DAY	Day of the week as a 9-character string	SUNDAY, MONDAY

Masks for date format strings (continued)

Mask	Description	Example values
DY	Day of the week as a 3-character string	SUN, MON, TUE, WED
J	Julian date (number of days from 1/1/4712 BC)	2443711
TH	String ST, ND, RD, or TH after a number	1ST, 2ND, 3RD, 4TH

Examples of date format strings:

Date format string	Sample results
‘DD-MON-YYYY’	02-JAN-1997
‘month/dd/yyyy’	january/02/1997
‘DAY,MONTH,DD’	THURSDAY,JANUARY,02

Time format strings

A time format string can contain any of the following masks along with other characters, such as colons (:) and periods. StorHouse/RM replaces the mask with values and retains any other characters.

Masks for time format strings

Mask	Description
AM	String of AM for time values before noon
PM	String of PM for time values after noon
A.M.	String of A.M. for time values before noon
P.M.	String of P.M. for time values after noon
HH	Hour as a 2-digit number in the range 00 to 24
HH12	Hour as a 2-digit number in the range 01 to 12
HH24	Hour as a 2-digit number in the range 00 to 24

Masks for time format strings (continued)

Mask	Description
MI	Minutes as a 2-digit number in the range 00 to 59
SS	Seconds as a 2-digit number in the range 00 to 62
SSSSS	Seconds from midnight in the range 00000 to 86399
MLS	Milliseconds as a 3-digit number in the range 000 to 999
MSC	Microseconds as a 6-digit number in the range 000000 to 999999

Examples of time format strings:

Time format string	Sample results
’HH PM’	14 PM
’HH12’	2
’HH24 pm’	14 pm
‘HH:MI:SS PM’	10:21:00 PM
‘HH24:MI:MSC pm’	22:21:000025 pm
‘HH12:MI:SSSSS’	10:21:80460
‘HH.MM’	10.21

Operators

StorHouse SQL supports the following operators:

- Set
- Logical
- Arithmetic
- Comparison
- Concatenation

Set operators (UNION and UNION ALL)

Set operators combine separate queries. StorHouse SQL supports the following set operators:

Set operator	Description
UNION	Merges the output of two or more queries into a single set of rows and columns, excluding duplicate rows from the output.
UNION ALL	Merges the output of two or more queries into a single set of rows and columns, including duplicate rows in the output.

Note the following:

- If a query contains a set operator, the ORDER BY clause can specify only the column position, not the column name. See page 4-92 for an example.
- To derive a result set using set operators, the tables being combined must have the same number of columns and the data types must be compatible for corresponding columns.
- Set operators are not valid for subqueries.

The following table describes the valid combinations of operand columns and the result data types for the UNION set operator.

UNION data type combinations

If one operand is	And the other is	The result data type is
BINARY(x)	BINARY(y)	BINARY(z) where z=MAX(x,y)
BINARY(x)	VARBINARY(y)	VARBINARY(z) where z=MAX(x,y)
BINARY(x)	BLOB(y)	BLOB(z) where z=MAX(x.y)
BLOB(x)	BLOB(y)	BLOB(z) where z=MAX(x,y)

UNION data type combinations (continued)

If one operand is	And the other is	The result data type is
BLOB(x)	VARBINARY(y)	BLOB(z) where z=MAX(x,y)
CHAR(x)	CHAR(y)	CHAR(z) where z=MAX(x,y)
CHAR(x)	VARCHAR(y)	VARCHAR(z) where z=MAX(x,y)
CHAR(x)	CLOB(y)	CLOB(z) where z=MAX(x,y)
CLOB(x)	CLOB(y)	CLOB(z) where z=MAX(x.y)
CLOB(x)	VARCHAR(y)	CLOB(z) where z=MAX(x,y)
DATE	DATE	DATE
DOUBLE PRECISION	Any numeric type	DOUBLE PRECISION
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
NUMERIC	NUMERIC	NUMERIC
NUMERIC	INTEGER	NUMERIC
NUMERIC	SMALLINT	NUMERIC
REAL	REAL	REAL
SMALLINT	SMALLINT	SMALLINT
TIME	TIME	TIME
TIMESTAMP	TIMESTAMP	TIMESTAMP
VARBINARY(x)	VARBINARY(y)	VARBINARY(z) where z=MAX(x,y)
VARBINARY(x)	BLOB(y)	BLOB(z) where z=MAX(x,y)
VARCHAR(x)	VARCHAR(y)	VARCHAR(z) where z=MAX(x,y)
VARCHAR(x)	CLOB(y)	CLOB(z) where z=MAX(x,y)

Logical operators (AND, OR, NOT)

Logical (or *boolean*) *operators* relate one or more predicates and produce a true, false, or unknown value. StorHouse SQL supports the following logical operators:

Logical operator	Description
AND	Takes two predicates as arguments and evaluates them as true only if both predicates are true.
OR	Takes two predicates as arguments and evaluates them as true if either one of the predicates is true.
NOT	Takes a predicate as its argument and changes its value from false to true or true to false.

Note the following:

- You can restrict a search to preclude a result from a query by using the NOT operator. For example, NOT(STATE='NY') excludes New York state from the query result.
- When you include multiple logical operators, StorHouse/RM evaluates conditions within parentheses first. If you don't include parentheses, then StorHouse/RM evaluates NOT conditions before AND and AND conditions before OR. StorHouse/RM optimizes conditions with the same logical operator, for instance, AND and AND.

The results for the OR and AND operators are shown in the following table.

OR and AND operator results table

Predicate 1	Predicate 2	Predicate 1 and 2	Predicate 1 or 2
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	Unknown	Unknown	Unknown

Arithmetic operators (+, -, *, /, unary)

You can use *arithmetic operators* in expressions to add, subtract, multiply, and divide numeric values. You can also perform addition and subtraction with columns defined with DATE or TIME data types. For instance, you could select all employees from an employee table where the system date minus the hire date is greater than 365 days.

StorHouse SQL supports the following arithmetic operators:

Arithmetic operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
+ (unary plus)	Do not change the sign
- (unary minus)	Negate a positive numeric value

The result precision and scale for arithmetic operations involving DECIMAL values are as follows.

result = val_1 <operation> val_2

where:

- result has precision p and scale s, and
- val_1 has precision p1 and scale s1, and
- val_2 has precision p2 and scale s2, and
- m is defined as (p - s), the integer part of the value

Result precision and scale for operations on DECIMAL values

Operation	Result	
+ or -	m = MAX (m1, m2) + 1	s = MAX (s1, s2)
*	m = m1 + m2	s = s1 + s2
/	m = m1 + s2	s = MAX (MIN (m2, 6) + s1 + 1, 8)

Note that m and s are limited to 31. The p (or s if division) is further reduced if needed to limit p to 31.

Comparison operators (=, <>, <, >, <=, >=)

Comparison operators compare two expressions. If the value of either expression is NULL or no value is returned, the result is unknown. StorHouse SQL supports the following comparison operators. In the table, a represents one expression and b represents the second.

Comparison operator	Example	Means
=	a=b	a is equal to b.
<>	a<>b	a is not equal to b.
<	a<b	a is less than b.
>	a>b	a is greater than b.
<=	a<=b	a is less than or equal to b.
>=	a>=b	a is greater than or equal to b.

Note: The allowable comparison operators for expressions of BLOB and CLOB data types are = and <>.

Concatenation operator (||)

The *concatenation operator* || is a synonym for the CONCAT scalar function. This operator links two string arguments to form a string expression. The following table identifies the result data type and length of the concatenated operands. See "CONCAT" on page 6-16 for more information about concatenation.

Concatenation result data types

If one operand is	And the other is	The result data type is
CHAR,VARCHAR(A)	CHAR,VARCHAR(B)	VARCHAR(MIN(A+B,32705)
CLOB(A)	CLOB(B)	CLOB(MIN(A+B,2G))
BLOB(A)	BLOB(B)	BLOB(MIN(A+B,2G))

NULL values

A *NULL value* is a special value distinct from all non-NULL values. It indicates that a value is unknown, missing, or not applicable. A column allows NULL values unless you specify the NOT NULL constraint. All data types accommodate NULL values. Note the following:

- You can test for NULL values by using the NULL predicate.
  - If the value of an expression is NULL, the result is true.
  - If the value of an expression is NOT NULL, the result is false.
  - If you specify the NOT keyword, the result of an expression is reversed.
- For determining duplicate rows, two NULL values are considered equal.
- All NULL values within a grouping column (the result from a GROUP BY clause) are considered equal.

Result columns allow NULL values if they are derived from:

- The COUNT function
- A column that allows NULL values
- A view column in an outer select list derived from an arithmetic expression
- An arithmetic expression in an outer select list
- An arithmetic expression that allows NULL values

- A scalar function or string expression that allows NULL values
- A result of a UNION if at least one of the corresponding items in the select list is nullable

Special registers

Special registers are storage areas for information that can be referenced in SQL statements. StorHouse supports three special registers:

- USER
- SYSDATE
- SYSTIME

USER special register

The *USER special register* contains the account ID of the user of the current transaction. The data type is VARCHAR(33). You can place USER wherever a string literal is allowed. You can also specify USER as a default value for a column.

For example, if the INFO table has an INFO_OWNER column identifying the account ID that entered the data, the following SELECT statement returns from INFO all ADDR data where INFO_OWNER is the account ID of the current user:

```
SELECT ADDR
FROM INFO
WHERE INFO_OWNER = USER
```

SYSDATE special register

The *SYSDATE special register* contains the current (system) date. The data type is DATE. You can place SYSDATE wherever a string literal is allowed. For example,

if the INFO table has a CREATE_DATE column identifying the date the data was entered, the following SELECT statement would return from table INFO all ADDR data archived on the current (system) date:

```
SELECT ADDR
FROM INFO
WHERE CREATE_DATE= SYSDATE
```

SYSTIME special register

The *SYSTIME special register* contains the current (system) time. You can place SYSTIME in a CREATE TABLE statement as a default column definition for a column with a data type of TIME.

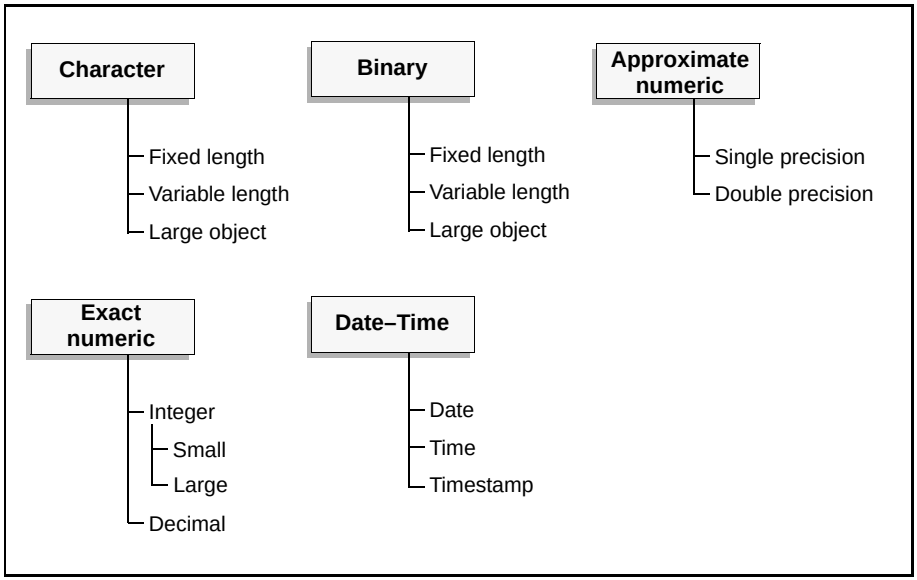
Chapter 3

StorHouse data types

This chapter describes the data types you can use to define columns, describe input data for loading and result data for unloading, and declare variables in programs. It also describes data type conversions that occur in StorHouse.

About StorHouse data types

A *data type* defines the properties of a data value. There are several categories of data types.



In StorHouse, you use data types to:

- Define columns in user tables. These data types are called *database data types*. See the following section for more information about database data types.
- Describe input data that you are loading. These data types are called *loader data types*. See page 3-17 for more information about loader data types.
- Describe output data that you are unloading. These data types are called *unloader data types*. See page 3-17 for more information about unloader data types.
- Declare host variables and indicator variables in programs. These data types are called *host language data types*. See page 3-21 for more information about host language data types.

Database data types

When you create a user table with a CREATE TABLE statement, you define columns. When you define a column, you assign a data type to it. This data type, called *database data type*, specifies the type of value that you can store in that column and indicates the maximum value length.

One consideration when defining a column is the type of data the column will contain. Types include character, numeric (exact or approximate), binary, and date–time. Sometimes you don't have a choice. For instance, when data values contain letters and special characters, a character data type is the only option. But if data values contain only digits, you can specify a character data type or a numeric data type. A determining factor might be whether you need to perform arithmetic operations with those data values (see page 3-14).

Another consideration when choosing a data type is whether to assign a variable-length or fixed-length data type. For variable-length data types, StorHouse/RM stores the data values in the actual number of bytes needed, plus two more bytes

per data value to hold the length of the value. So a data value consisting of 10 bytes is stored in 12 bytes. For fixed-length data types, StorHouse/RM reserves the same amount of space for all values, even those with fewer bytes than allocated. So if you specify a fixed length of 10, data values consisting of 2 bytes are stored in 10 bytes, those consisting of 6 bytes are stored in 10 bytes, and so on. When there's a large variation in lengths of data values in a column, consider a variable-length data type.

Descriptions of database data types

This section describes the StorHouse database data types.

BINARY

The BINARY data type defines bit data as a fixed-length array of bytes.

BINARY specifications	
Category:	Binary, fixed
Format:	BINARY[(length)]
Stored length:	Value of (length)
Length range:	1 through 256, inclusive
Default length:	1
Example:	BINARY(5) means that data values in a column must be bit data with a maximum length of 5 bytes. StorHouse/RM reserves 5 bytes for all values in all rows of this column, and it pads any remaining bytes with binary zeros for those values with fewer than 5 bytes.
Considerations:	■ StorHouse/RM does not translate BINARY data from host to host. ■ Do not declare host variables as the BINARY data type. Instead use an SQLDA and the DESCRIBE statement. Refer to the *StorHouse ESQL Manual* for more information.

BLOB The BLOB data type defines data as a variable-length array of bytes. A BLOB is primarily used to hold large, unformatted data such as images, audio, and video.

BLOB specifications	
Category:	Binary, large object
Format:	BLOB[(length [K|M|G])] BINARY LARGE OBJECT[(length [K|M|G])]
Stored length:	In-line LOB: actual size of the BLOB data plus 4 bytes Out-of-line: actual size of the BLOB data in the LOB subsegment file plus 22 bytes for the object identifier (OID) in the table data file
Length range:	1 byte to 2147483638 bytes K (kilobyte) – 1 to 2097152 M (megabyte) – 1 to 2048 G (gigabyte) – 1 or 2
Default length:	2 G
Examples:	■ BLOB(1 G) means that values in the column must be binary data with a maximum length of 1G. ■ BLOB (without a length) uses the default maximum length, which is 2 G.
Consideration:	BLOB is a valid data type for most of the functions that support BINARY and VARBINARY data types.

CHARACTER

The CHARACTER data type defines data as a fixed-length array of characters including letters, numbers, spaces, and special characters.

CHARACTER specifications	
Category:	Character, fixed
Format:	CHARACTER[(length)] or CHAR[(length)]
Stored length:	Value of (length)
Length range:	1 through 256, inclusive
Default length:	1
Examples:	■ CHAR(5) means that data values in a column must be character data with a maximum length of 5 bytes. StorHouse/RM reserves 5 bytes for all values in all rows of this column, and it pads (to the right) any remaining bytes with blanks for those values with fewer than 5 bytes. For instance, StorHouse/RM stores the value ‘ABC’ as ‘ABC ’. ■ CHAR (without a length) uses the default length, which is 1.
Considerations:	■ StorHouse/RM can translate CHAR data between different hosts. For example, an IBM mainframe host and a UNIX host can both retrieve the same CHAR values translated by StorHouse/RM. ■ CHAR data values typically contain non-numeric data. Consider a numeric data type (such as NUMERIC, INTEGER, or SMALLINT) for data values on which you want to perform arithmetic functions. ■ You can use the USER special register with CHAR columns.

CLOB The CLOB data type defines character data as a variable-length array of bytes.

CLOB specifications	
Category:	Character, large object
Format:	CLOB[(length [K|M|G])] CHARACTER LARGE OBJECT[(length [K|M|G])]
Stored length:	In-line LOB: actual size of the CLOB data plus 4 bytes Out-of-line LOB: actual size of the CLOB data in the LOB subsegment file plus 22 bytes for the object identifier (OID) in the table data file
Length range:	1 byte to 2147483638 bytes K (kilobytes) – 1 to 2097152 M (megabytes) – 1 to 2048 G (gigabytes) – 1 or 2
Default length:	2 G
Examples:	■ CLOB(10 M) means that values in the column must be character data with a maximum length of 10 megabytes. ■ CLOB (without a length) uses the default maximum length, which is 2 G.
Considerations:	■ StorHouse/RM can translate CLOB data between different hosts. For example, an IBM mainframe host and a UNIX host can both retrieve the same CLOB values translated by StorHouse/RM. ■ CLOB is a valid data type for most of the functions that support CHAR and VARCHAR data types.

DATE

The DATE data type defines a date value with three parts: month, day, and year.

DATE specifications	
Category:	Date–time
Format:	DATE
Range for parts:	■ Month – 1 through 12 ■ Day – 1 through 28, 30, or 31 (depending on the month) ■ Year – 0001 through 9999
Stored length:	4 bytes
Considerations:	■ You can perform scalar functions with DATE values. For example, in a SELECT statement you can select rows with a specific day in a month (1 through 31) by using the DAYOFMONTH function or on a specific day of the week (Monday through Sunday) by using the DAYOFWEEK function, and so on. ■ You can use the SYSDATE special register with DATE columns. ■ Gregorian calendar rules are applied over the entire DATE range.

DOUBLE PRECISION

The DOUBLE PRECISION (synonym FLOAT) data type defines data as a double precision floating-point number.

DOUBLE PRECISION or FLOAT specifications	
Category:	Approximate numeric, double precision
Format:	DOUBLE PRECISION or FLOAT
Floating-point:	$\pm$ fraction * base$^{\pm \text{ exponent}}$ where sign is 1 bit, fraction is 52 bits, and exponent is 11 bits
Stored length:	8 bytes (64 bits)
Considerations:	DOUBLE PRECISION is suitable for scientific numbers that can be calculated approximately. This data type defines much more precise data (for example, more significant digits in the fraction) than REAL.

INTEGER

The INTEGER data type defines data as a signed large integer.

INTEGER specifications	
Category:	Exact numeric, large integer
Format:	INTEGER or INT
Range of values:	-2147483648 through +2147483647, inclusive
Stored length:	4 bytes
Considerations:	Arithmetic operations and sort comparisons are more efficient using INTEGER instead of NUMERIC or DOUBLE PRECISION data types.

NUMERIC or DECIMAL

The NUMERIC (synonym DECIMAL) data type defines data as a decimal number with a precision (P) and a scale (S).

NUMERIC or DECIMAL specifications	
Category:	Exact numeric, decimal
Format:	NUMERIC(P[,S]) or DECIMAL(P[,S]) ■ P is the total number of digits, from 1 to 31. ■ S is the number of digits to the right of the decimal point, from 0 to P.
Range of values:	$1\text{-}10^{31}$ to $10^{31} - 1$
Default values:	Precision is 31 and scale is 0
Stored length:	(P/2)+1
Example:	NUMERIC(12,2) defines data values with 12 total digits, 10 digits before the decimal point and 2 digits after the decimal point.
Considerations:	You can perform functions such as ABS (absolute value) and AVG (average) with NUMERIC data.

REAL The REAL data type defines data as a single precision floating-point number.

REAL specifications	
Category:	Approximate numeric, single precision
Format:	REAL
Floating point:	± fraction * base$^{\pm \text{ exponent}}$ where sign is 1 bit, fraction is 23 bits, and exponent is 8 bits
Stored length:	4 bytes (32 bits)
Considerations:	REAL is suitable for scientific numbers that can be calculated approximately.

SMALLINT The SMALLINT data type defines data as a signed small integer.

SMALLINT specifications	
Category:	Exact numeric, small integer
Format:	SMALLINT
Range of values:	-32768 through +32767, inclusive
Stored length:	2 bytes
Considerations:	You may want to use the SMALLINT data type when column values are less than 1,000. In this case, you can save space by using SMALLINT instead of INTEGER.

TIME The TIME data type defines a time value with four parts: hour, minutes, seconds, and milliseconds.

TIME specifications	
Category:	Date–time
Format:	TIME
Range for parts:	■ Hour – 00 through 24 ■ Minutes – 00 through 59 ■ Seconds – 00 through 62 ■ Milliseconds – 000 through 999 If the hour is 24, the minutes, seconds, and milliseconds are 00.
Stored length:	4 bytes
Considerations:	■ You can perform scalar functions with TIME values. For example, you can select rows for a specific hour (00 through 24) with the HOUR function, or for a specific minute (00 through 59) with the MINUTE function, and so on. ■ You can use the SYSTIME special register with TIME columns. ■ You can use seconds values greater than 60 to indicate leap seconds; however, time calculations don't consider leap seconds.

TIMESTAMP

The TIMESTAMP data type defines a date and time value as a month, day, year, hour, minutes, seconds, and microseconds.

TIMESTAMP specifications	
Category:	Date–time
Format:	TIMESTAMP
Range for parts:	■ Month – 1 through 12 ■ Day – 1 through 28, 30, or 31 (depending on the month) ■ Year – 0001 through 9999 ■ Hour – 00 through 24 ■ Minutes – 00 through 59 ■ Seconds – 00 through 62 ■ Microseconds – 000000 through 999999
Stored length:	8 bytes

VARBINARY

The VARBINARY data type defines bit data as a variable-length array of bytes.

VARBINARY specifications	
Category:	Binary, variable
Format:	VARBINARY[(length)]
Maximum length:	Value of (length)
Stored length:	Actual length of data plus 2 bytes
Length range:	1 through 32705, inclusive
Default length:	1
Example:	VARBINARY(25) means that data values in a column must be bit data and cannot be longer than 25 bytes. StorHouse/RM reserves two additional bytes for each data value in this column. For instance, a data value composed of 25 bytes is stored in 27 bytes.
Considerations:	■ StorHouse/RM does not translate VARBINARY data from host to host. ■ You can create an index on a VARBINARY column, but the maximum length is 256. ■ Do not declare host variables as the VARBINARY data type. Instead, use an SQLDA and the DESCRIBE statement. Refer to the *StorHouse ESQL Manual* for more information. ■ StorHouse/RM automatically compresses any VARBINARY column defined with a maximum length greater than 4096. The compression algorithm uses consecutive blank and duplicate character elimination. Sometimes the length of a compressed block of data can actually exceed the length of the original uncompressed data block. The StorHouse/RM compression algorithm automatically recognizes when this occurs and bypasses compression for such blocks. The effect is overall improvement in compression effectiveness.

VARCHAR

The VARCHAR data type defines data as a variable-length array of characters including letters, numbers, spaces, and special characters.

VARCHAR specifications	
Category:	Character, variable
Format:	VARCHAR[(length)]
Maximum length:	Value of (length)
Stored length:	Actual length of data plus 2 bytes
Length range:	1 through 32705, inclusive
Default length:	1
Example:	VARCHAR(16) means that a data value must consist of character data and cannot be longer than 16 bytes. StorHouse/RM reserves two additional bytes for each data value in this column. For instance, a data value composed of 16 bytes is actually stored in 18 bytes.
Considerations:	■ The maximum length of a VARCHAR data value that you can index is 256 bytes. ■ StorHouse/RM can translate VARCHAR data between different hosts. For example, an IBM mainframe host and a UNIX host can both retrieve the same VARCHAR values translated by StorHouse/RM. ■ Do not declare host variables as the VARCHAR data type. Instead, use an SQLDA and the DESCRIBE statement. Refer to the *StorHouse ESQL Manual* for more information. ■ StorHouse/RM automatically compresses any VARCHAR column defined with a maximum length greater than 4096. The compression algorithm uses consecutive blank and duplicate character elimination. Sometimes the length of a compressed block of data can actually exceed the length of the original uncompressed data block. The StorHouse/RM compression algorithm automatically recognizes when this occurs and bypasses compression for such blocks. The effect is overall improvement in compression effectiveness.

Database data types and functions

Another consideration when choosing a data type for a column is whether you can use scalar or aggregate functions on those columns. For instance, although a column defined as CHARACTER can contain digits, you cannot perform arithmetic operations on character types; but you can perform arithmetic operations on a numeric types, such as SMALLINT or INTEGER.

The following table lists the database data types and identifies the functions you can perform on columns defined as that type. See Chapter 6 for more information about StorHouse functions.

Allowable functions by database data type

Data type	StorHouse functions		
BINARY	BIT_LENGTH CONCAT NVL POSITION TRIM	BLOB COUNT OCTET_LENGTH TO_CHAR	CHAR_LENGTH LENGTH OVERLAY TO_HEX
BLOB	BIT_LENGTH COUNT LTRIM OVERLAY SUBSTR	CHAR_LENGTH INSTR NVL POSITION TO_HEX	CONCAT LENGTH OCTET_LENGTH RTRIM TRIM
CHARACTER	ASCII CONCAT INSTR LOWER MAX OVERLAY RTRIM TO_DATE TRANSLATE	BLOB COUNT LEAST LPAD MIN POSITION SUBSTR TO_NUMBER TRIM	CLOB INITCAP LENGTH LTRIM NVL RPAD TO_CHAR TO_TIME UPPER

Allowable functions by database data type (continued)

Data type	StorHouse functions		
CLOB	ASCII CONCAT INSTR LPAD OCTET_LENGTH RPAD TO_CHAR TO_TIME UPPER	BIT_LENGTH COUNT LENGTH LTRIM OVERLAY RTRIM TO_DATE TRANSLATE	CHAR_LENGTH INITCAP LOWER NVL POSITION SUBSTR TO_NUMBER TRIM
DATE	ADD_MONTHS DAYOFWEEK GREATEST MIN NVL WEEK	COUNT DAYOFYEAR LEAST NEXT_DAY QUARTER YEAR	DAYOFMONTH DECODE MAX MONTHS_BETWEEN TO_CHAR
DOUBLE PRECISION or FLOAT	ABS DECODE MAX SUM	AVG GREATEST MIN TO_CHAR	COUNT LEAST NVL
INTEGER	ABS COUNT LEAST NVL	AVG DECODE MAX SUM	CHR GREATEST MIN TO_CHAR
NUMERIC or DECIMAL	ABS DECODE MAX SUM	AVG GREATEST MIN TO_CHAR	COUNT LEAST NVL
REAL	ABS DECODE MAX SUM	AVG GREATEST MIN TO_CHAR	COUNT LEAST NVL
SMALLINT	ABS COUNT LEAST NVL	AVG DECODE MAX SUM	CHR GREATEST MIN TO_CHAR

Allowable functions by database data type (continued)

Data type	StorHouse functions		
TIME	HOUR TO_CHAR	MINUTE	SECOND
TIMESTAMP	HOUR TO_CHAR	MINUTE	SECOND
VARBINARY	BIT_LENGTH COUNT OVERLAY TO_HEX	CHAR_LENGTH NVL POSITION TRIM	CONCAT OCTET_LENGTH TO_CHAR
VARCHAR	ASCII CLOB INITCAP LENGTH LTRIM NVL POSITION SUBSTR TO_NUMBER TRIM	BIT_LENGTH CONCAT INSTR LOWER MAX OCTET_LENGTH RPAD TO_CHAR TO_TIME UPPER	CHAR_LENGTH COUNT LEAST LPAD MIN OVERLAY RTRIM TO_DATE TRANSLATE

StorHouse and local database data types

The StorHouse database data types may differ slightly from the data types used in a local database. For example, the following table compares DB2 data types to StorHouse database data types. Remember that when you write StorHouse SQL statements, you use StorHouse data types instead of those supported by a local database.

DB2 equivalent to StorHouse database data types

StorHouse data type	DB2 equivalent
BINARY	CHAR FOR BIT DATA
BLOB	BLOB
CHAR	CHAR
CLOB	CLOB
DATE	DATE IN USA FORMAT
DOUBLE PRECISION	DOUBLE PRECISION or FLOAT(53)
FLOAT	FLOAT(53)
INTEGER	INTEGER
NUMERIC	NUMERIC
REAL	REAL
SMALLINT	SMALLINT
TIME	TIME IN USA FORMAT
TIMESTAMP	TIMESTAMP
VARBINARY	VARCHAR FOR BIT DATA
VARCHAR	VARCHAR

Loader and unloader data types

When you load data, you describe the data fields in your input data records. Typically, a *data field* corresponds to a column in a user table, and an *input data record* corresponds to a row. When you describe a data field, you provide the data type and optionally the length in a LOAD DATA statement. This data type is called a *loader data type*.

When you unload data, you select data from a StorHouse database and receive the result data in a sequential file. The FileTek FTP Data Unloader uses the data

type to convert the result data. You must use one of the data types that are supported by the FileTek FTP Data Unloader. The names of the unloader data types are the same as the loader data types. Refer to the *FileTek FTP Unloader Manual* for definitions of the data type specifications.

List of loader and unloader data types

The data types that you use in a LOAD DATA and UNLOAD statement differ slightly from the database data types that you use in a CREATE TABLE statement. There are some additional abbreviations and data types that are valid only for defining input and result data. For example, you can define input data fields with data type INTEGER EXTERNAL; you cannot define a column in a user table with data type INTEGER EXTERNAL. You can use some synonyms such as RAW to define input data fields; you cannot define columns in a user table with data type RAW. The following table lists the loader and unloader data types and corresponding synonyms or abbreviations.

Note: The BLOB, BLOB_FILE, CLOB, and CLOB_FILE data types do not apply to the FileTek MVS Data Loader utility.

Loader and unloader data types

Data type	Synonym or abbreviation
BINARY	BYTE, RAW, CHAR CHARSET 65535
BINARY EXTERNAL	BYTE EXTERNAL, RAW EXTERNAL
BLOB	–
BLOB_FILE	–
CHARACTER	CHAR
CLOB	–
CLOB_FILE	–
DATE EXTERNAL	DATE
DECIMAL	NUMERIC, DEC

Loader and unloader data types (continued)

Data type	Synonym or abbreviation
DECIMAL EXTERNAL	NUMERIC EXTERNAL, DEC EXTERNAL
DOUBLE PRECISION	FLOAT(22 to 53), DOUBLE
FLOAT	FLOAT(1 to 21), REAL
FLOAT EXTERNAL	–
INTEGER	INT
INTEGER EXTERNAL	INT EXTERNAL
SMALLINT	–
TIME EXTERNAL	–
TIMESTAMP EXTERNAL	–
VARBINARY	VARBYTE, VARRAW, VARCHAR CHARSET 65535
VARCHAR	–

Date and time formats of input and result data

When you load date and time data, those values must be in a format recognized by StorHouse. When you unload data, you can specify a format mask for the

result data. The following table identifies the valid formats of date and time input and result data.

Formats for loading and unloading date and time data

Loader data type	Input or result data format
DATE EXTERNAL	YYYY-M[M]-D[D] M[M]/D[D]/YYYY D[D].M[M].YYYY
TIME EXTERNAL	H[H]:M[M] H[H].M[M] H[H]:M[M] AM|PM H[H]:M[M]:S[S] H[H]:M[M]:S[S].CCC H[H].M[M].S[S].CCC
TIMESTAMP EXTERNAL	YYYY-M[M]-D[D]-H[H].M[M].S[S].[CCCCCC] M[M]/D[D]/YYYY H[H]:M[M]:S[S].[CCCCCC] M[M]/D[D]/YYYY H[H].M[M].S[S].[CCCCCC]

The year value (YYYY) can range from 0001 to 9999. For example, the value 0099 refers to year 99. The value 1999 refers to year 1999.

The default formats for unloading data are as follows:

- DATE EXTERNAL – MM/DD/YYYY
- TIME EXTERNAL – H24:MI:SS.MLS
- TIMESTAMP EXTERNAL – MM/DD/YYYY HH24:MI:SS.MLS

A FileTek data loader accepts a TIME data field with no milliseconds (MLS) field. The FileTek unloader produces the MLS field only if the milliseconds part of the TIME value being unloaded is nonzero. If you specify a length that is less than 12, the default mask does not contain MLS.

Host language data types

If you use StorHouse ESQL to submit SQL statements in your programs, then you declare host variables and indicator variables. You can declare variables with StorHouse data types or host language data types. A *host language data type* is a data type supported by the host language, which for ESQL is C or C++.

- To declare host variables with a host language data type, you can use the C data types char, double, long, and short. StorHouse ESQL also supports guaranteed-size types for C type int and long. See the table on page 3-23 for a list of these types.

- To declare host variables with a StorHouse data type, you can use all database data types except BINARY, VARBINARY, and VARCHAR.

- To declare indicator variables, you can use the StorHouse data type SMALLINT or the C data type short. You can also declare a new data type based on SMALLINT or short.

- To declare the following StorHouse data types, you use C language structures: BLOB, BLOB_FILE, CLOB, CLOB_FILE, DATE, NUMERIC (and synonym DECIMAL), TIME, and TIMESTAMP.

The following table maps StorHouse data types to C language data types and structures. Refer to the *StorHouse ESQL Manual* for more information about host language data types.

C language equivalent to StorHouse data types

StorHouse data type	C language data type	C language structure
BINARY[1]	unsigned char	–
BLOB	–	tpe_blob_t
BLOB_FILE	–	tpe_blob_file_t
BLOB_LOCATOR	int32_t	–
CHARACTER	char	–
CLOB	–	tpe_clob_t
CLOB_FILE		tpe_clob_file_t
CLOB_LOCATOR	int32_t	
DATE	–	tpe_date_t
DOUBLE PRECISION	double	–
INTEGER	int32_t	–
NUMERIC	–	tpe_num_t
REAL	float[2]	–
SMALLINT	short	–
TIME	–	tpe_time_t
TIMESTAMP	–	tpe_timestamp_t
VARBINARY[1]	unsigned char	–
VARCHAR[1]	char	–

[1] Do not declare host variables as BINARY, VARBINARY, or VARCHAR data types. Instead, use an SQLDA and the DESCRIBE statement.

[2] In C language, the float data type represents a single precision floating-point number. However, ESQL interprets the word “float” as the database data type DOUBLE (synonym FLOAT), which indicates a double precision floating-point number. To declare a variable as a single precision floating-point number, use database data type REAL.

Depending on the platform, the C type int may be 16 or 32 bits and the C type long may be 32 or 64 bits. StorHouse ESQL makes the following guaranteed-size types available to applications. You can use these type definitions in place of the built-in C types to ensure portability to new platforms.

Guaranteed-size types for C type int and long

Type	Description
int64_t	64-bit signed integer
uint64_t	64-bit unsigned integer
int32_t	32-bit signed integer
uint32_t	32-bit unsigned integer
int16_t	16-bit signed integer
uint16_t	16-bit unsigned integer
int8_t	8-bit signed integer
uint8_t	8-bit unsigned integer

Data type conversion

Several data type conversions occur in StorHouse, including conversion of loader data types, unloader data types, literals, and functions.

Conversion of loader data types

A loader data type doesn't have to match a database data type. For example, you can define an input data field as BINARY and then load those values into columns defined as BINARY, BLOB, CHARACTER, CLOB, VARBINARY, or VARCHAR. StorHouse/RM automatically performs the necessary conversion; but you need to ensure that the data types are compatible and that the input data conforms to the format of the column's data type.

For instance, your load will fail if you try to load data values defined as SMALLINT into a column defined as CHARACTER. These data types are not compatible. Your load will also fail if the input data doesn't conform to the format of the column's data type. For example, although you can load CHARACTER data values into a SMALLINT column, those values must consist of digits (such as 123), not letters (such as ABC).

You also need to ensure that the length of your input data doesn't exceed the length of the target column. StorHouse/RM trims any data values that exceed the length of a column's data type. For example, if you define a column in a user table as CHARACTER(30) and define input data fields as CHARACTER(35), then StorHouse/RM would trim any bytes from data values with more than 30 bytes.

StorHouse/RM also rescales DECIMAL and NUMERIC columns when necessary. For example, if the input data is DECIMAL(7,3) but the column is NUMERIC(7,2), StorHouse/RM rescales and stores the input data as NUMERIC (7,2).

The following table identifies the allowable data type conversions for loading data into StorHouse. In the table:

- The *Input type* is the data type of the input data (loader data type).
- The *Target type* is the data type of the table being loaded (CREATE TABLE data type).

Synonyms are not listed but are supported. For example, BYTE, a synonym for BINARY, has the same conversions as BINARY.

Data type conversions for loading data

Input type	Target type													
	BINARY	BLOB	CHARACTER	CLOB	DATE	DOUBLE PRECISION	INTEGER	NUMERIC (DECIMAL)	REAL	SMALLINT	TIME	TIMESTAMP	VARBINARY	VARCHAR
BINARY	X	X	X	X									X	X
BINARY EXTERNAL	X	X	X	X		X	X		X	X			X	X
BLOB		X		X										
BLOB_FILE		X												
CHARACTER	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CLOB		X		X										
CLOB_FILE				X										
DATE					X									
DECIMAL							X	X		X				
DECIMAL EXTERNAL							X	X		X				
DOUBLE						X	X		X	X				
FLOAT(REAL)						X	X		X	X				
FLOAT EXTERNAL						X	X	X	X	X				
INTEGER							X			X				
INTEGER EXTERNAL							X			X				
SMALLINT							X			X				
TIME EXTERNAL											X			
TIMESTAMP EXTERNAL												X		

Data type conversions for loading data (continued)

Input type	Target type													
	BINARY	BLOB	CHARACTER	CLOB	DATE	DOUBLE PRECISION	INTEGER	NUMERIC (DECIMAL)	REAL	SMALLINT	TIME	TIMESTAMP	VARBINARY	VARCHAR
VARBINARY	X	X	X	X									X	X
VARCHAR	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Conversion of unloader data types

StorHouse/RM automatically performs necessary data type conversions when you unload data, but you must ensure that the result data types are compatible with the column data types. The following table summarizes the data type conversions for unloading data from StorHouse. In the table:

- The *Result type* is the data type of the result data (the unloader data type).
- The *Source type* is the data type of the expression being unloaded.

Synonyms are not listed but are supported.

Data type conversions for unloading data

Result type	Source type													
	BINARY	BLOB	CHARACTER	CLOB	DATE	DOUBLE PRECISION	INTEGER	NUMERIC (DECIMAL)	REAL	SMALLINT	TIME	TIMESTAMP	VARBINARY	VARCHAR
BINARY	X	X	X	X									X	X
BINARY EXTERNAL	X	X	X	X		X	X		X	X			X	X
BLOB		X		X										
BLOB_FILE		X												
CHARACTER	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CLOB		X		X										
CLOB_FILE				X										
DATE EXTERNAL					X							X		
DECIMAL								X						
DECIMAL EXTERNAL								X						
DOUBLE						X			X					
FLOAT(REAL)						X			X					
FLOAT EXTERNAL						X			X					
INTEGER							X			X				
INTEGER EXTERNAL							X			X				
SMALLINT							X			X				
TIME EXTERNAL											X	X		
TIMESTAMP EXTERNAL												X		
VARBINARY	X	X	X	X									X	X
VARCHAR	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Conversion of literals

StorHouse/RM is flexible in that you can use different literal types to specify a value. For instance, you can use a character literal (quoted string) to specify a value in a column defined as a CHARACTER, CLOB, DATE, INTEGER, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, or VARCHAR data type. The literal must conform to the format of the column data type. For example, if you specify a character literal for a NUMERIC data value, then that character literal should contain decimal digits and a decimal point.

Here's an example. Assume you defined a column named COL1 with data type INTEGER. You can specify a literal for an INTEGER data value in a variety of formats, including an integer literal:

```
SELECT *
FROM MYTABLE
WHERE COL1 = 1024
```

or a character literal (quoted string):

```
SELECT *
FROM MYTABLE
WHERE COL1 = '1024'
```

In the first query, no conversion occurs because the integer literal is the same format as the INTEGER column. In the second query, however, StorHouse/RM converts the character literal to the data type of the column (INTEGER). So the string value 1 - 0 - 2 - 4 becomes the integer value 1024 (one thousand twenty four).

The following table identifies the types of literals you can use with database data types.

Literals you can use with database data types

Literal	Result data type													
	BINARY	BLOB	CHAR	CLOB	DATE	DOUBLE	INTEGER	NUMERIC	REAL	SMALLINT	TIME	TIMESTAMP	VARBINARY	VARCHAR
Character			X	X	X		X	X	X	X	X	X		X
Decimal						X	X	X	X	X				
Floating point			X			X	X	X	X	X				X
Hexadecimal	X	X											X	
Integer			X			X	X	X	X	X				X

Conversion of function data types

When you perform a function, the result data type may differ from the data type of the specified expression. For example, the LENGTH function returns the length (INTEGER data type) of a character expression (CHARACTER data type). The TO_NUMBER function converts a CHARACTER data value to a NUMERIC data value.

The following table identifies the result data type of the StorHouse functions.

Result data type of StorHouse functions

Function	Result data type
ABS	Depends on expression
ADD_MONTHS	DATE
ASCII	SMALLINT

Result data type of StorHouse functions (continued)

Function	Result data type
AVG	Depends on expression
BIT_LENGTH	DECIMAL
BLOB	BLOB
CHAR_LENGTH	INTEGER
CHR	CHAR
CLOB	CLOB
CONCAT	Depends on expressions
COUNT	INTEGER
DAYOFMONTH	SMALLINT
DAYOFWEEK	SMALLINT
DAYOFYEAR	SMALLINT
DAYS	INTEGER
DECODE	Same as first match expression
GREATEST	Same as first expression
HOUR	SMALLINT
INITCAP	Same as expression
INSTR	INTEGER
LAST_DAY	DATE
LEAST	Same as first expression
LENGTH	INTEGER
LOWER	Same as expression
LPAD	Same as expression
LTRIM	Same as expression
MAX	Same as expression

Result data type of StorHouse functions (continued)

Function	Result data type
MIN	Same as expression
MINUTE	SMALLINT
MONTH	SMALLINT
MONTHS_BETWEEN	INTEGER
NEXT_DAY	DATE
NVL	Same as first expression
OCTET_LENGTH	INTEGER
OVERLAY	Depends on first expression
POSITION	INTEGER
QUARTER	SMALLINT
RPAD	Same as expression
RTRIM	Same as expression
SECOND	SMALLINT
SUBSTR	Same as first expression
SUM	DECIMAL
TO_CHAR	VARCHAR or CLOB, depending on expression
TO_DATE	DATE
TO_HEX	CHAR or CLOB
TO_NUMBER	NUMERIC
TO_TIME	TIME
TRANSLATE	Same as first expression
TRIM	Depends on first expression
UPPER	Same as expression

Result data type of StorHouse functions (continued)

Function	Result data type
WEEK	SMALLINT
YEAR	SMALLINT

Chapter

4

StorHouse SQL statements

This chapter contains formats and examples of StorHouse SQL statements.

About StorHouse SQL statements

StorHouse SQL *statements* are instructions used to manage StorHouse databases. Statements consist of one or more parts that form a *clause*. Clauses begin with a keyword followed by arguments. *Arguments* complete or modify the meaning of the keyword (for example, WHERE CUSTOMER='SMITH').

Statements generally act on components in a database. A *component* is a named database structure. Components include tables, views, columns, user tablespaces, synonyms, and indexes.

The use of these statements is controlled by database and database component privileges assigned by the StorHouse system or database administrator. You must have certain privileges to use many of these statements. The privileges required to use a statement are listed in the description for each statement.

List of StorHouse SQL statements

The following table contains an alphabetical list and a description of the StorHouse SQL statements. All StorHouse SQL statements can be static, but only some can be both static and dynamic. The Dynamic column means the statement can be dynamically prepared.

List of StorHouse SQL statements

SQL statement	Description	Dynamic	Page
ALTER TABLE SPACE	Changes user tablespace parameters	Yes	4-7
BEGIN and END DECLARE SECTION	Declares variables, types, and arrays	No	4-11
CLOSE	Closes an open cursor	No	4-19
COMMIT WORK	Terminates a transaction, making changes to the database permanent	No	4-20
CONNECT	Opens a connection with a database	No	4-22
CREATE INDEX	Creates a hash, value, or range index for a user table	Yes	4-24
CREATE SYNONYM	Creates a synonym for a table, view, or synonym	Yes	4-27
CREATE TABLE	Creates an empty user table	Yes	4-29
CREATE TABLE SPACE	Creates a user tablespace	Yes	4-35
CREATE VIEW	Creates a view for a table or view	Yes	4-40
DECLARE	Assigns a cursor name and associates a cursor with a query	No	4-42
DELETE	Removes one or more rows from a system table or system table view	Yes	4-44
DESCRIBE	Provides information about input or output host variables	No	4-46

List of StorHouse SQL statements (continued)

SQL statement	Description	Dynamic	Page
DISCONNECT	Terminates a connection between a program and a database	No	4-49
DROP INDEX	Removes an index	Yes	4-50
DROP SYNONYM	Removes a synonym	Yes	4-52
DROP TABLE	Removes a user table	Yes	4-53
DROP TABLE SPACE	Removes a user tablespace	Yes	4-54
DROP VIEW	Removes a view	Yes	4-56
EXECUTE	Executes a prepared (dynamic) non-SELECT SQL statement	No	4-57
EXECUTE IMMEDIATE	Prepares and executes an SQL statement represented as a statement string or host variable	No	4-60
FETCH	Retrieves the result set for queries that return multiple rows	No	4-62
FREE LOCATOR	Releases the server storage used by a locator variable	No	4-66
GRANT	Grants database or database component privileges to StorHouse accounts	Yes	4-67
INSERT	Inserts new rows into a system table or system table view	Yes	4-70
OPEN	Executes a query for a cursor and identifies the rows in the result set	No	4-72
PREPARE	Parses an SQL statement for syntax errors and assigns an identifier to the statement	No	4-75
REVOKE	Removes database or database component privileges from a StorHouse account	Yes	4-77

List of StorHouse SQL statements (continued)

SQL statement	Description	Dynamic	Page
ROLLBACK WORK	Cancels the current transaction and rolls back any changes performed during the transaction	No	4-80
SELECT	Retrieves information from one or more tables	Yes	4-82
SET CONNECTION	Makes the named connection the current one	No	4-105
UPDATE	Changes some or all values in a system table or system table view	Yes	4-106
VALUES INTO	Manipulates values and expressions previously selected using locator variables	Yes	4-108
WHENEVER	Specifies an action for NOT FOUND, SQLERROR, and SQLWARNING runtime exceptions	No	4-109

Categories of SQL

There are four categories of StorHouse SQL statements:

- *Data Definition Language* (DDL) statements create, alter, and drop database components and grant and revoke privileges. StorHouse/RM implicitly commits a transaction before and after every DDL statement. This means that DDL statements are atomic; you cannot roll them back.
- *Data Manipulation Language* (DML) statements query and manipulate data.
- *Transaction Control* statements manage changes to a database.
- *ESQL* statements, declarative and executable, can be included in a program.

The following table lists the specific statements in each category.

Categories of StorHouse SQL statements

Category	SQL statement
Data Definition Language (DDL)	ALTER TABLE SPACE CREATE INDEX CREATE SYNONYM CREATE TABLE CREATE TABLE SPACE DROP INDEX DROP SYNONYM DROP TABLE DROP TABLE SPACE DROP VIEW GRANT REVOKE
Data Manipulation Language (DML)	DELETE INSERT SELECT UPDATE VALUES INTO

Categories of StorHouse SQL statements (continued)

Category	SQL statement
Transaction Control	COMMIT WORK ROLLBACK WORK
ESQL	BEGIN and END DECLARE SECTION CLOSE CONNECT DECLARE DESCRIBE DISCONNECT EXECUTE EXECUTE IMMEDIATE FETCH FREE LOCATOR OPEN PREPARE SET CONNECTION WHENEVER

ALTER TABLE SPACE

ALTER TABLE SPACE changes the storage specifications of an existing user tablespace. Any new segments are stored according to the new specifications. Existing segments are stored according to the old specifications. You must have DBA privilege to alter a user tablespace. Note the following:

- You cannot use a ROLLBACK WORK statement to undo an ALTER TABLE SPACE statement.
- At least one SUBSPACE clause with one argument is required. StorHouse/RM does not check for a specific argument as part of its required syntax.
- Enclose all of the subspace clauses in one set of parentheses. Separate subspace clauses with commas.
- You must delimit volume set and file set names that do not follow SQL identifier naming conventions.
- TABLE SPACE can be one or two words (TABLE SPACE or TABLESPACE).

Format

```
ALTER TABLE SPACE tablespace_name
( SUBSPACE subspace_number param_value [ param_value]...
[,SUBSPACE subspace_number param_value [ param_value]... ]... )
```

where param_value is any of the following:

```
VSET vset_name
FSET fset_name
OBJECT_TYPE type_value
ATF atf_value
VTF vtf_value
EDC edc_value
```

GROUP group_name
MAX_EXT_SIZE size_in_megabytes
HOLD number_of_days
HOLD_SPECIAL number_of_days

Argument	Description
tablespace_name	(required) Name of the user tablespace.
subspace_number	(required) Number assigned to the subspace you are changing.
param_value	(one required) Parameter(s) to change for this subspace.
VSET vset_name	Name of the volume set to contain components in this subspace.
FSET fset_name	Name of the file set to contain components in this subspace.
OBJECT_TYPE type_value	Type of component to be stored in the subspace. Valid values are: ■ blank – (default) Allow all components. ■ T – Allow table data only. ■ H – Allow hash indexes only. ■ V – Allow value indexes only. ■ L – Allow LOB data only.
ATF atf_value	Access Time Factor for components in this subspace. Valid values are: ■ 0 – (default) Use the value of the StorHouse ATF system parameter. ■ 1 – Short access time is very important. ■ 2 – Short access time is moderately important. ■ 3 – Short access time is minimally important.

Argument	Description
VTF vtf_value	Vulnerability Time Factor for components in this subspace. Valid values are: ■ DEFAULT – (default) Use the value of the StorHouse VTF system parameter. ■ NOW – Write the file to the performance buffer first and then copy it immediately to its file set. ■ NEXT – Write the file to the performance buffer and copy it to its file set during the next StorHouse write-back operation. ■ DIRECT – Bypass the performance buffer and write the file directly to its file set. (DF and map extents, however, are always copied to the performance buffer as well as to the file set.)
EDC edc_value	Error Detection Code (EDC) flag indicating whether the StorHouse error detection capability is to be used for components in this subspace. Valid values are: ■ D – (default) Use the value of the StorHouse EDC system parameter. ■ Y – Use EDC. ■ N – Do not use EDC.
GROUP group_name	Name of the StorHouse file access group to contain components in this subspace. If you include the GROUP keyword with an empty delimited name ("") or an all-blank delimited name or all blanks, StorHouse/RM uses the default group STH.
MAX_EXT_SIZE size_in_megabytes	Maximum size of extents in this subspace. Valid values are: ■ 0 – (default) 100 MB for LOB subsegment files or use the value of the applicable StorHouse system parameter for the component type: SQL_MAX_EXT_DATA for table data files SQL_MAX_EXT_HASH for hash index files SQL_MAX_EXT_VAL for value index files ■ 1 to the maximum surface size for the VSET. Refer to the *StorHouse Concepts and Facilities Manual* for details on extent size considerations.

Argument	Description
HOLD number_of_days	Number of days to keep data extents in the performance buffer. Valid values are: ■ 0 – (default) 0 days for LOB subsegment files or use the value of the applicable StorHouse system parameter for the component type: SQL_HOLD_DATA for table data files SQL_HOLD_INDX for hash and value index files ■ 1 to 32767
HOLD_SPECIAL number_of_days	Number of days to keep DF and map extents of table data files, index files, and LOB subsegment files in the performance buffer. Valid values are: ■ 0 – (default) Use the value of the SQL_HOLD_SPECIAL StorHouse system parameter. ■ 1 to 32767

Examples

- The following ALTER TABLE SPACE statement changes the volume set and file set names assigned to SUBSPACE 1 in the BILLINGTABLE user tablespace.

  ```
  ALTER TABLE SPACE BILLINGTABLE
  (SUBSPACE 1 VSET FEB2000T FSET FEB2000T)
  ```

- The following ALTER TABLE SPACE statement changes the ATF value for SUBSPACE 1, the MAX_EXT_SIZE for SUBSPACE 2, and the number of days to hold special extents in the performance buffer for SUBSPACE 3 in the BILLINGJAN user tablespace.

  ```
  ALTER TABLE SPACE BILLINGJAN
  ( SUBSPACE 1 ATF 1 ,
  SUBSPACE 2 MAX_EXT_SIZE 400,
  SUBSPACE 3 HOLD_SPECIAL 32767 )
  ```

BEGIN and END DECLARE SECTION

BEGIN DECLARE SECTION and END DECLARE SECTION mark the beginning and end of a Declare Section in an ESQL program. A Declare Section contains declaration statements in ESQL or C language formats. You can declare:

- Host and indicator variables as certain host language or StorHouse data types
- Variables as host arrays
- New data types with the characteristics of host language or StorHouse data types; then declare variables and arrays using the newly defined data type

Note the following:

- Your program can contain more than one Declare Section.
- A Declare Section cannot reference names declared in typedef statements.
- A Declare Section must be located before other ESQL constructs. Only C statements can precede a Declare Section.

Format

```
EXEC SQL BEGIN DECLARE SECTION
    declaration_statements
EXEC SQL END DECLARE SECTION
```

where declaration_statements include variable_declarations, array_declarations, and type_declarations:

variable_declarations	
ESQL format	variable_name IS OF TYPE type_category
C format	type_category variable_name

array_declarations	
ESQL format	variable_name IS AN ARRAY OF type_name WITH SIZE constant_id
C format	(non-char array) host_language_type_name variable_name [constant_id] (char array) host_language_type_name variable_name [constant_id] [length]

type_declarations	
ESQL format	TYPE new_type_name IS OF TYPE type_category TYPE new_type_name IS AN ARRAY OF element_type_name WITH SIZE constant_id

Declaring host and indicator variables

You can declare host variables (including locator variables and file reference variables) and indicator variables in ESQL or C language formats. Note the following:

- You can declare host variables with all of the data types listed in the table on page 4-13.
- You cannot declare host variables as BINARY, VARBINARY, and VARCHAR data types. Instead, use an SQLDA and the DESCRIBE statement.
- You can declare indicator variables as C type short or database type SMALLINT.
- C type float typically represents a single precision floating-point number; however, ESQL interprets float as a synonym for DOUBLE (a double precision floating-point number). To declare a variable as a single-precision floating-point number, use the StorHouse REAL data type.

ESQL format

variable_name IS OF TYPE type_category

where type_category is:

{ host_language_type | storhouse_type }

C language format

type_category variable_name

where type_category is:

{ host_language_type | storhouse_type }

Argument	Description
variable_name	(required) Name of the host variable being declared.
host_language_type	(required for specifying a host language type) Name of the host language type. ESQL supports the following C types: [unsigned] char double [unsigned] short [unsigned] long In place of C type int or long, you can use a guaranteed-size type listed in the table on page 3-23.
storhouse_type	(required for specifying a StorHouse data type) Name of the StorHouse data type. BLOB BLOB_FILE BLOB_LOCATOR CHARACTER CLOB CLOB_FILE CLOB_LOCATOR DATE DECIMAL DOUBLE FLOAT INTEGER NUMERIC REAL SMALLINT TIME TIMESTAMP See the table on page 3-22 for the mapping of StorHouse data types to C language types and structures.

Examples

- The following example uses the ESQL format to declare the host variable customer_no as the StorHouse type INTEGER:

```
EXEC SQL BEGIN DECLARE SECTION ;
    customer_no IS OF TYPE INTEGER ;
EXEC SQL END DECLARE SECTION ;
```

- The following example uses the C language format to declare the host variable customer_no as the C language type long:

```
EXEC SQL BEGIN DECLARE SECTION ;
    long customer_no ;
EXEC SQL END DECLARE SECTION ;
```

- The following example uses the C language format to declare the locator variable hv_prod_locator as StorHouse type CLOB_LOCATOR and the host variable hv_product as StorHouse type CLOB:

```
EXEC SQL BEGIN DECLARE SECTION ;
    CLOB_LOCATOR hv_prod_locator ;
    CLOB(2M) hv_product ;
EXEC SQL END DECLARE SECTION ;
```

Declaring arrays

An *array* is a group of data items or elements assigned to one variable name. In a Declare Section, you declare host and indicator variables as arrays, identifying the data type and setting the size of the array. You can declare an array in ESQL format or in C language format.

ESQL format to declare an array

variable_name IS AN ARRAY OF type_name WITH SIZE constant_id

Argument	Description
variable_name	(required) Name of the array being declared.
type_name	(required) Data type assigned to variable_name.
constant_id	(required) Number of elements in variable_name.

C language format to declare a non-char array

host_language_type_name variable_name [constant_id]

Argument	Description
host_language_type_name	(required) Data type assigned to variable_name.
variable_name	(required) Name of the array being declared.
constant_id	(required) Number of elements in variable_name.

C language format to declare a char array

host_language_type_name variable_name [constant_id] [length]

Argument	Description
host_language_type_name	(required) Data type assigned to variable_name.
variable_name	(required) Name of the array being declared.
constant_id	(required) Number of elements in variable_name.
length	(required) Length of each char element in variable_name.

Examples

- The following example uses the ESQL format to declare the array arrayexample. This array is of type long and has 12 elements:

```
EXEC SQL BEGIN DECLARE SECTION ;
    arrayexample IS AN ARRAY OF long OF SIZE 12 ;
EXEC SQL END DECLARE SECTION ;
```

- The following example uses the C language format to declare the non-char array arrayexample. This array is of type long and has 12 elements:

```
EXEC SQL BEGIN DECLARE SECTION ;
    long arrayexample [12] ;
EXEC SQL END DECLARE SECTION ;
```

- The following example uses the C language format to declare the char array myarray. There are 10 elements in the array, and each element is 2 characters long:

```
EXEC SQL BEGIN DECLARE SECTION ;
    char myarray [10] [2];
EXEC SQL END DECLARE SECTION ;
```

Declaring type definitions for variables and arrays

You can define a new data type with the same characteristics as an existing host language or StorHouse data type. Then you can declare host variables and arrays using the new data type.

ESQL format to declare a new data type for a variable

TYPE new_type_name IS OF TYPE type_category

where type_category is defined as:

host_language_type | storhouse_type

Argument	Description
new_type_name	(required) Name of the new type being declared.
host_language_type	(required for specifying a host language type) Name of the host language type. ESQL supports the following C types: [unsigned] char double [unsigned] short [unsigned] long In place of C type int or long, you can use a guaranteed-size type listed in the table on page 3-23.
storhouse_type	(required for specifying a StorHouse data type) Name of the StorHouse data type. BLOB BLOB_FILE BLOB_LOCATOR CHARACTER CLOB CLOB_FILE CLOB_LOCATOR DATE DECIMAL DOUBLE FLOAT INTEGER NUMERIC REAL SMALLINT TIME TIMESTAMP See the table on page 3-22 for the mapping of StorHouse data types to C language types and structures.

ESQL format to declare a new data type for an array

TYPE new_type_name IS AN ARRAY OF element_type_name
WITH SIZE constant_id

Argument	Description
new_type_name	(required) Name of the new type you are declaring.
element_type_name	(required) Host language type or StorHouse type that appears in the array.
constant_id	(required) Size of each element in the array.

Examples

- The following example defines a new data type called customer_no with the characteristics of C type unsigned long. It then declares the host variable input_v as the type customer_no:

```
EXEC SQL BEGIN DECLARE SECTION ;
    TYPE customer_no IS OF TYPE unsigned long ;
    customer_no input_v ;
EXEC SQL END DECLARE SECTION ;
```

- The following example defines a new data type called array_type with the same characteristics as the C type char. The new data type array_type represents a char array where each element contains 10 characters. The example then declares the array my_array using the new data type array_type. There are 30 elements in my_array.

```
EXEC SQL BEGIN DECLARE SECTION ;
    TYPE array_type IS AN ARRAY OF char WITH SIZE 10 ;
    my_array IS AN ARRAY OF array_type WITH SIZE 30 ;
EXEC SQL END DECLARE SECTION ;
```

CLOSE

CLOSE sets the state of an open cursor to closed. StorHouse/RM automatically closes a cursor when a transaction is committed, but it's good practice to close a cursor explicitly. Once a cursor is closed, you cannot perform FETCH operations on the cursor.

Format

```
EXEC SQL
    CLOSE cursor_name
```

Argument	Description
cursor_name	(required) Name of the open cursor that you want to close.

Example

The following example closes the open cursor named cust_cur:

```
EXEC SQL
    CLOSE cust_cur ;
```

COMMIT WORK

COMMIT WORK terminates a transaction by making all changes to the database during the transaction permanent. It also releases all held locks, open cursors, server storage used by locator variables, and prepared statements for the transaction. Once you commit a transaction, you cannot cancel the changes. Should the system fail, StorHouse/RM automatically cancels a transaction and rolls back the database to its initial state.

Note the following:

- In StorHouse/RM, DDL statement execution is atomic and permanent. The software always performs an implicit COMMIT WORK before and after every DDL statement. You cannot roll back changes to a StorHouse database after issuing DDL statements.

- Non-DDL statements, like PREPARE and SELECT, place locks on tables until the transaction is committed. Therefore, you must explicitly issue a COMMIT WORK statement after each non-DDL statement to release those locks. You can cancel the effect of non-DDL processing before committing a transaction by issuing the ROLLBACK WORK statement.

Format

```
EXEC SQL
    COMMIT WORK
```

Examples

- The following example commits the change to the sysadm.syssmusers system table:

```
EXEC SQL
    UPDATE sysadm.syssmusers
    SET default_ts = :def_tbspace
    WHERE accountid = :acct_id ;

EXEC SQL
    COMMIT WORK ;
```

- The following example commits the SELECT statement and releases the locks that were held on emptable.

```
EXEC SQL
    SELECT empname
    FROM emptable ;

EXEC SQL
    COMMIT WORK ;
```

CONNECT

CONNECT opens a connection between a program and a database. A program can connect to multiple databases but execute SQL statements in only one database at a time. A program can open up to 10 connections at a time. The SQL_SESSIONS system parameter sets the total number of connections allowed system-wide.

Caution: The USING clause, which supplies the StorHouse account password, is optional. However, if you omit the USING clause in a CONNECT statement in an ESQL program, StorHouse/RM will prompt stdin for the password. If stdin is not a terminal, the CONNECT will fail.

Format

```
EXEC SQL
    CONNECT TO database_name
    [ AS connection_name ]
    [ [ USER account_id [USING :host_variable_name ] ] ]
```

Argument	Description
database_name	(required) Name of the database, expressed as a character literal or a host variable. When connecting to a remote database, you must specify a connect string in the following format: filetek:T:remote_host_name:database_name
connection_name	(optional) Name of the connection, expressed as a character literal or host variable.
account_id	(required with the USER clause) StorHouse account ID, expressed as a character literal or a host variable.
:host_variable_name	(required with the USING clause) StorHouse account password expressed as a host variable. A literal is not accepted.

Example

The following example connects in remote mode to a database called salesdb:

```
EXEC SQL
    CONNECT TO ‘filetek:T:remotehost:salesdb’ AS ‘conn_2’
    USER ‘jack’ USING :pword ;
```

In this example:

- filetek:T:remotehost:salesdb is the connect string
- filetek is a constant
- T is a constant
- remotehost is the name of the remote machine containing salesdb
- salesdb is the StorHouse database name
- conn_2 is the connection name
- jack is the StorHouse account ID
- :pword is the host variable for the StorHouse account password

CREATE INDEX

CREATE INDEX creates an index for the specified user table. StorHouse supports three types of indexes: value, hash, and range. You must have DBA or RESOURCE privilege, own the user table, or have INDEX privilege on the user table to create an index. Note the following:

- You cannot use a ROLLBACK WORK statement to undo a CREATE INDEX statement.
- You can create indexes before or after a user table is loaded.
- If a table contains data, you must create a deferred index by including the DEFERRED keyword. An error occurs if you omit the DEFERRED keyword and the table contains data. After creating the deferred index, use a FileTek data loader to load the index for existing segments.
- If you omit the VALUE, HASH, or RANGE keywords, StorHouse/RM uses the value of the SQL_IDX_TYPE system parameter to determine which type of index to create.
- The maximum number of columns you can specify in a single CREATE INDEX statement is 16.
- If you omit the TABLE SPACE clause, StorHouse/RM assigns the index to the same user tablespace as the table, that is, the user tablespace specified on the CREATE TABLE statement.

Format

```
CREATE [VALUE | HASH | RANGE] INDEX index_name
ON [owner.] table_name ( column_name [, column_name]... )
[ DEFERRED ]
[ TABLE SPACE tablespace_name ]
```

Argument	Description
VALUE	(optional) Type of index that contains an ordered list of all row values for an indexed column and maps the row values to their corresponding row data in the table data file. Best used for queries based on a range of values (for example, greater than or less than).
HASH	(optional) Type of index that uses hash values for each index entry. Best used for queries based on equalities.
RANGE	(optional) Type of index that contains the highest and lowest values in each segment. Only the appropriate segment is searched. Best used for queries on tables with multiple segments.
index_name	(required) Name of the index you are creating. This name must be unique within the database.
owner.	(optional) Account ID of the owner of the user table.
table_name	(required) Name of the user table for which you are creating the index.
column_name	(required) Name of the column in the index. If you specify more than one column, StorHouse/RM creates a *compound index*. The column can be any database data type except BLOB or CLOB.
DEFERRED	(optional) Type of index created after a table has been loaded. If the table contains data, you must include this keyword. You can create a deferred index for all index types: value, hash, and range. After creating the index, use a FileTek data loader to load the deferred index (by using a LOAD INDEX statement) for existing segments.
tablespace_name	(optional) Name of the user tablespace to contain the index. TABLE SPACE can be one or two words. If you omit this clause, the default is the same user tablespace as the user table.

Examples

- The following CREATE INDEX statement creates a HASH index named ORDINDEX using the ORDER_NO column in the ORDERS table. The index is stored in the same user tablespace as the table (no TABLE SPACE clause).

  ```
  CREATE HASH INDEX ORDINDEX
  ON ORDERS (ORDER_NO)
  ```

- The following CREATE INDEX statement creates a compound VALUE index named CUSTOMER_ORDERS using the ORDER_NO and CUSTOMER_NAME columns in the ORDERS table. The index is assigned to the BILLING user tablespace.

  ```
  CREATE VALUE INDEX CUSTOMER_ORDERS
  ON ORDERS (ORDER_NO, CUSTOMER_NAME)
  TABLE SPACE BILLING
  ```

- The following CREATE INDEX statement creates a compound VALUE index named ORDERS2000 for the ORDERS table. This statement contains the DEFERRED keyword to create a deferred index.

  ```
  CREATE VALUE INDEX ORDERS2000
  ON ORDERS (ORDER_NO, ORDER_DATE, CUSTOMER_NAME)
  DEFERRED
  TABLE SPACE BILLING
  ```

CREATE SYNONYM

CREATE SYNONYM creates a synonym (another name) for the specified table, view, or synonym. You can create private synonyms for your own use or if authorized, public synonyms for all StorHouse accounts. You must have DBA or RESOURCE privilege to use CREATE SYNONYM. You must have DBA privilege to use CREATE PUBLIC SYNONYM.

Note: You cannot use a ROLLBACK WORK statement to undo a CREATE SYNONYM statement.

Format

CREATE [PUBLIC] SYNONYM synonym_name
FOR [owner.]{table_name | view_name | synonym}

Argument	Description
PUBLIC	(optional) Indicates that any account can refer to this synonym without having to specify the owner's account ID. If you omit PUBLIC, StorHouse/RM creates a private synonym. To reference a private synonym owned by another account, you must start that synonym name with the owner's account ID.
synonym_name	(required) Name of the synonym you are creating.
owner.	(optional) Account ID of the owner of the table, view, or synonym to be associated with this synonym.
table_name	(required if you are creating a synonym for a table) Name of the table to be associated with this synonym.
view_name	(required if you are creating a synonym for a view) Name of the view to be associated with this synonym.
synonym	(required if you are creating a synonym for another synonym) Name of the synonym to be associated with this synonym.

Example

The following CREATE SYNONYM statement creates the PUBLIC synonym PUBLIC_SUPPLIERS for the SUPPLIERS table, which is owned by SMITH.

```
CREATE PUBLIC SYNONYM PUBLIC_SUPPLIERS
FOR SMITH.SUPPLIERS
```

CREATE TABLE

CREATE TABLE defines a user table in the current database. You must specify at least one column definition. CREATE TABLE requires DBA or RESOURCE privilege or DBA privilege to create a user table for another owner. Note the following:

- Only a FileTek data loader can insert rows into a user table. You cannot use INSERT, UPDATE, or DELETE statements to add, change, or remove rows in user tables.
- You cannot use a ROLLBACK WORK statement to undo a CREATE TABLE statement.
- Table names (and other database user components like indexes, views, synonyms, and user tablespaces) may not start with SYS, for instance, SYS_STARTUP or SYSSERVICE. The SYS prefix is reserved for system tables.

Format

```
CREATE TABLE [owner.]table_name
({column_definition} [,column_definition]... )
[TABLE SPACE tablespace_name]
```

where column_definition is defined as:

```
column_name column_type [DEFAULT default_definition]
[column_constraints] [lob_options]
```

and where lob_options is defined as:

```
[INLINE [(length [K])] | NOT INLINE]
[STORE WITH column_name]
[TABLE SPACE tablespace_name]
```

Argument	Description
owner.	(optional) Account ID of the table owner. If you omit the owner, StorHouse/RM makes you the owner of the user table. You must have DBA privilege to specify another account as owner of the table.
table_name	(required) Name of the user table. The fully qualified table name is the combined owner name and user table name. This combination must be unique from other tables, views, and synonyms in the current database.
column_name	(required) Name of the column. A column name must be unique within a user table.
column_type	(required for each column definition) Data type of the column. The lengths are optional and if omitted, the default length is used.
BINARY(length)	Fixed-length array of bit data with a length from 1 to 256 bytes.
BLOB(length [K|M|G]) or BINARY LARGE OBJECT (length [K|M|G])	Variable-length array of bytes with a length from: ■ 1 to 2147483638 bytes ■ 1 to 2097152 K ■ 1 to 2048 M ■ 1 or 2 G
CHAR(length) or CHARACTER(length)	Fixed-length array of characters with a length from 1 to 256.
CLOB(length [K|M|G]) or CHARACTER LARGE OBJECT(length [K|M|G])	Variable-length array of characters measured in bytes with a length from: ■ 1 to 2147483638 bytes ■ 1 to 2097152 K ■ 1 to 2048 M ■ 1 or 2 G
DATE	Date value representing a month, day, and year.
DOUBLE PRECISION	Double precision floating-point number.

Argument	Description
INTEGER	Large integer value ranging from -2147483648 to +2147483647.
NUMERIC(P[,S]) or DECIMAL(P[,S])	Signed decimal number with a precision (P) and scale (S). P can range from 1 to 31, with a default of 31. S can range from 0 to P, with a default of 0.
REAL	Single precision floating-point number.
SMALLINT	Small integer value ranging from -32768 to +32767.
TIME	Time value representing an hour, minutes, seconds, and milliseconds.
TIMESTAMP	Date and time combination representing a month, day, year, hour, minutes, seconds, and microseconds.
VARBINARY(length)	Variable-length array of bit data ranging in length from 1 to 32705 bytes inclusive.
VARCHAR(length)	Variable-length array of characters ranging in length from 1 to 32705 characters inclusive. The maximum length for an indexed VARCHAR column is 256.
DEFAULT default_definition	(optional) Default value for the column. During a load, the column takes the default value if no value is supplied.
literal	Specific binary, numeric, or character constant.
NULL	NULL value. If you omit DEFAULT, the default is NULL.
SYSDATE	Current date. SYSDATE is valid for DATE columns only.
SYSTIME	Current time. SYSTIME is valid for TIME columns only.
USER	Account ID performing the load. USER is valid for CHAR and VARCHAR columns that have a minimum length of 12.

Argument	Description
column_constraints	(optional) Column-level constraints that StorHouse applies to the column during a load. The load returns an error if the value being loaded does not satisfy the column constraint.
NOT NULL	Null values are not allowed in this column.
CHECK (srch_cnd)	CHECK is not currently implemented.
TABLE SPACE tablespace_name	(optional) Name of the user tablespace to contain this user table. TABLE SPACE can be one or two words. If you omit this clause, StorHouse/RM uses your account default user tablespace. If you do not have an account default user tablespace, StorHouse/RM uses the database default user tablespace.
lob_options	(optional for BLOB and CLOB column definitions) Column-level options for a LOB.
INLINE [(length [K])]	Option to store a column's LOB values with the table data and to set a maximum length for in-line data. The default maximum length is 32 K. ■ If you omit the length, StorHouse/RM stores a LOB value with the table data when the row size does not exceed 32705 bytes. ■ If you specify a length, StorHouse/RM stores a LOB value with the table data as long as the value does not exceed the in-line length limit and the row length limit. For instance, if the in-line length limit for a LOB column is 10 K, StorHouse/RM stores any LOB values that are 10 K or less and that fit in the row.
NOT INLINE	Option to always store a column's LOB values in separate LOB subsegment files from the table data file.

Argument	Description
STORE WITH column_name	Name of a different LOB column in which to store this column's LOB values. This option enables you to store the values of multiple LOB columns together in the same LOB subsegment file(s).
TABLE SPACE tablespace_name	Name of the user tablespace to contain this column's LOB values. If you omit this clause, StorHouse/RM stores the column's LOB values in the same user tablespace as the user table.

Examples

- The following CREATE TABLE statement creates a user table named SUPPLIER_ITEM with three INTEGER columns: SUPP_NO, ITEM_NO, and QTY. The SUPP_NO column cannot contain NULL values. The table is associated with a user tablespace named SUPL0395.

```
CREATE TABLE SUPPLIER_ITEM
(SUPP_NO INTEGER NOT NULL,
ITEM_NO INTEGER,
QTY INTEGER)
TABLE SPACE SUPL0395
```

- The following CREATE TABLE statement creates a user table named EMP with two INTEGER columns (EMPNO, DEPTNO) and a DATE column (JOIN_DATE) in user tablespace EMPL.

```
CREATE TABLE EMP
(EMPNO INTEGER,
DEPTNO INTEGER DEFAULT 10,
JOIN_DATE DATE DEFAULT NULL),
TABLE SPACE EMPL
```

Default values are assigned for two columns.

– The default value for the DEPTNO column is 10. During a load, StorHouse/RM inserts a data value of 10 into the DEPTNO column when a value is not supplied.

– The default value of the JOIN_DATE column is NULL. StorHouse/RM inserts a NULL value into the JOIN_DATE column when a value is not supplied.

■ The following CREATE TABLE statement illustrates different LOB options.

```
CREATE TABLE LOB_EXAMPLE
(CHAR_COL CHAR(6),
INT_COL INTEGER,
FIRST_LOB CLOB STORE WITH OTHER_CLOB,
SECOND_LOB BLOB NOT INLINE,
OTHER_CLOB CLOB TABLESPACE XYZ )
TABLE SPACE ABC
```

– Values for the CHAR_COL and INT_COL are stored in a table data file in the ABC tablespace.

– If there are multiple LOB columns in a row that could be stored in-line (such as FIRST_LOB and OTHER_CLOB), the order of the LOB columns is important. In this example, FIRST_LOB column gets first priority for in-line storage.

– If the values for FIRST_LOB and/or OTHER_CLOB) fit in the row, they are stored in-line in the table data file in the ABC tablespace. But if they don't fit, the values for the FIRST_LOB and/or OTHER_CLOB columns are stored in the same LOB subsegment file(s) in the XYZ tablespace.

– Values for the SECOND_LOB column are stored in a different LOB subsegment file(s) (from the other LOBs and the table data file) in the ABC tablespace.

CREATE TABLE SPACE

CREATE TABLE SPACE creates a user tablespace in the current database. A user tablespace defines storage specifications for user table, index, and LOB data. You must have DBA privilege to create a user tablespace. Note the following:

- You cannot use a ROLLBACK WORK statement to undo a CREATE TABLE SPACE statement.
- At least one SUBSPACE clause is required.
- Enclose all of the subspace clauses in one set of parentheses. Separate subspace clauses with commas.
- If you omit the optional subspace parameters, the defaults are used.
- You must delimit volume set and file set names that do not follow SQL identifier naming conventions.
- TABLE SPACE can be one or two words (TABLESPACE or TABLE SPACE).

Format

```
CREATE TABLE SPACE tablespace_name
(SUBSPACE subspace_number VSET vset_name FSET fset_name
param_value [ param_value]...
[, SUBSPACE subspace_number VSET vset_name FSET fset_name
param_value [ param_value]... ]... )
```

where param_value is any of the following:

OBJECT_TYPE type_value
ATF atf_value
VTF vtf_value
EDC edc_value
GROUP group_name
MAX_EXT_SIZE size_in_megabytes
HOLD number_of_days
HOLD_SPECIAL number_of_days

Argument	Description
tablespace_name	(required) Name of the user tablespace. This name must be unique in a database.
subspace_number	(required) Number to assign to the subspace. This number can range from 0 to 2,147,483,647. Subspace numbers must be unique within a user tablespace.
VSET vset_name	(required) Name of the volume set to contain components in this subspace.
FSET fset_name	(required) Name of the file set to contain components in this subspace.
param_value	(optional) Storage parameter for this subspace.
OBJECT_TYPE type_value	(optional) Type of component to be stored in the subspace. Valid values are: ■ blank – (default) Allow all components. ■ T – Allow table data only. ■ H – Allow hash indexes only. ■ V – Allow value indexes only. ■ L – Allow LOB data only.

Argument	Description
ATF atf_value	(optional) Access Time Factor for components in this subspace. Valid values are: ■ 0 – (default) Use the value of the StorHouse ATF system parameter. ■ 1 – Short access time is very important. ■ 2 – Short access time is moderately important. ■ 3 – Short access time is minimally important.
VTF vtf_value	(optional) Vulnerability Time Factor for components in this subspace. Valid values are: ■ DEFAULT – (default) Use the value of the StorHouse VTF system parameter. ■ NOW – Write the file to the performance buffer first and then copy it immediately to its file set. ■ NEXT – Write the file to the performance buffer and copy it to its file set during the next StorHouse write-back operation. ■ DIRECT – Bypass the performance buffer and write the file directly to its file set. (DF and map extents, however, are always copied to the performance buffer as well as to the file set.)
EDC edc_value	(optional) Error Detection Code (EDC) flag indicating whether the StorHouse error detection capability is to be used for components in this subspace. Valid values are: ■ D – (default) Use the value of the StorHouse EDC system parameter. ■ Y – Use EDC. ■ N – Do not use EDC.
GROUP group_name	(optional) Name of the StorHouse file access group to contain components in this subspace. If you omit this parameter, the default group is STH. If you include the GROUP keyword with an empty delimited name ("") or an all-blank delimited name or all blanks, StorHouse/RM uses the default group STH.

Argument	Description
MAX_EXT_SIZE size_in_megabytes	(optional) Maximum size (in MB) of extents in this subspace. Valid values are: ■ 0 – (default) 100 MB for LOB subsegment files or use the value of the applicable system parameter for the file type: SQL_MAX_EXT_DATA for table data files SQL_MAX_EXT_HASH for hash index files SQL_MAX_EXT_VAL for value index files ■ 1 to the maximum surface size for the VSET. Refer to the *StorHouse Concepts and Facilities Manual* for details on extent size considerations.
HOLD number_of_days	(optional) Number of days to keep data extents in the performance buffer. Valid values are: ■ 0 – (default) 0 days for LOB subsegment files or use the value of the system parameter for the file type: SQL_HOLD_DATA for table data files SQL_HOLD_INDX for index files ■ 1 to 32767
HOLD_SPECIAL number_of_days	(optional) Number of days to keep DF and map extents in the performance buffer. Valid values are: ■ 0 – (default) Use the value of the SQL_HOLD_SPECIAL system parameter. ■ 1 to 32767

Examples

- The following CREATE TABLE SPACE statement creates a user tablespace called BILLINGTABLE with one subspace that uses all of the default values:

  ```
  CREATE TABLE SPACE BILLINGTABLE
  (SUBSPACE 1 VSET JAN2000T FSET JAN2000T)
  ```

- The following CREATE TABLE SPACE statement creates a user tablespace called BILLINGJAN with four subspaces, one for each component type.

  ```
  CREATE TABLE SPACE BILLINGJAN
  (SUBSPACE 1 VSET JAN2000T FSET JAN2000T OBJECT_TYPE T
  ATF 2 VTF NOW MAX_EXT_SIZE 400 HOLD 30 HOLD_SPECIAL 60,
  SUBSPACE 2 VSET JAN2000H FSET JAN2000H OBJECT_TYPE H
  ATF 1 VTF NEXT MAX_EXT_SIZE 800 HOLD 90 HOLD_SPECIAL 365,
  SUBSPACE 3 VSET JAN2000V FSET JAN2000V OBJECT_TYPE V
  ATF 1 VTF NEXT MAX_EXT_SIZE 500 HOLD 90 HOLD_SPECIAL 365
  SUBSPACE 4 VSET JAN2000L FSET JAN2000L OBJECT_TYPE L
  ATF 2 VTF NOW MAX_EXT_SIZE 800 HOLD 30 HOLD_SPECIAL 60)
  ```

CREATE VIEW

CREATE VIEW creates a view for one or more user tables, system tables, or views. To create a view, you must have the following privileges:

- DBA or RESOURCE privilege
- SELECT privilege on all references tables or views
- DBA privilege to create a view for another owner

Note the following rules for creating views:

- You cannot use a ROLLBACK WORK statement to undo a CREATE VIEW statement.
- The view must contain a single query. Subqueries are not allowed.
- The view can reference multiple tables and/or views.
- The query may not contain an ORDER BY clause.
- You cannot use DELETE, INSERT, or UPDATE statements with views derived from user tables. These statements are restricted to system tables and views based on system tables.

Format

CREATE VIEW [owner.]view_name
[(column_name [, column_name]...)]
AS query_expression

Argument	Description
owner.	(optional) Account ID of the owner of the view to be created.
view_name	(required) Name of the view to be created.
column_name	(optional) Name of the column to be included in the view. If omitted, the view uses the same column names specified in query_expression.
query_expression	(required) Query that defines and fills the new view.

Example

The following CREATE VIEW statement creates a six-column view named NE_CUSTOMERS. The view fills the columns with each row from the CUSTOMER table where STATE contains NH, MA, NY, or VT.

```
CREATE VIEW NE_CUSTOMERS AS
SELECT CUST_NO, NAME, STREET, CITY, STATE, ZIP
FROM CUSTOMER
WHERE STATE IN ('NH', 'MA', 'NY', 'VT')
```

DECLARE

DECLARE assigns a name to a cursor and associates a cursor with a static query or a prepared dynamic query. Place a DECLARE statement before any other SQL statement that references that cursor. ESQL cannot interpret a reference to a cursor that is not declared. Note the following:

- A cursor declared in one ESQL source file cannot be referenced in another ESQL source file.
- Cursor names must be unique within a file.
- The DECLARE statement and the following OPEN statement for the same cursor must occur within the same transaction definition.
- If the DECLARE statement contains references to host variables, then the associated OPEN statement must occur within the same scope as that of the referenced variables.

Format

```
EXEC SQL
    DECLARE cursor_name CURSOR FOR
    { query_expression | prepared_statement_name }
```

Argument	Description
cursor_name	(required) Unique name for the cursor.
query_expression	(required for static queries) SELECT statement you are associating with the cursor.
prepared_statement_name	(required for prepared dynamic queries) Name of the prepared statement assigned in a PREPARE statement.

Examples

- The following example declares the cursor cust_cur for a static query on the customer table:

```
EXEC SQL
    DECLARE cust_cur CURSOR FOR
    SELECT cust_no, name, street, city, state
    FROM customer ;
```

- The following example declares the cursor select_cursor for the prepared statement select_stmt represented by the string variable stmt_str:

```
EXEC SQL BEGIN DECLARE SECTION ;
    char stmt_str [256] ;
EXEC SQL END DECLARE SECTION ;
...
EXEC SQL
    PREPARE select_stmt FROM :stmt_str ;
    DECLARE select_cursor CURSOR for select_stmt ;
```

DELETE

DELETE removes zero, one, or more rows from the specified system table or system table view. You cannot delete rows from user tables or user table views. If you specify WHERE, only rows that satisfy the WHERE search condition are deleted. If you omit WHERE, then all rows of the specified system table or system table view are deleted.

You must have DBA privilege, DELETE privilege on the system table, or own the system table to use this statement. If the target is a view, DELETE privilege is required on the target base table referenced in the view definition. In StorHouse, the SYSADM account owns the system tables.

Format

DELETE FROM [owner.]{table_name | view_name}
[WHERE condition]

Argument	Description
owner.	(optional) Account ID of the owner of the system table or system table view.
table_name	(required if view_name is omitted) Name of the system table.
view_name	(required if table_name is omitted) Name of the system table view.
condition	(optional) Predicate that limits the range of rows deleted. If you omit the condition, StorHouse/RM deletes all rows in the specified system table or system table view. See Chapter 5, “StorHouse SQL Predicates,” for additional information about predicates.

Example

The following DELETE statement deletes the row that contains the account ID USER1 from the SYSSMUSERS system table.

```
DELETE FROM SYSADM.SYSSMUSERS
WHERE ACCOUNTID='USER1'
```

DESCRIBE

DESCRIBE provides information about input or output host variables in an SQL statement. The statement has two formats: DESCRIBE BIND VARIABLES and DESCRIBE SELECT LIST.

DESCRIBE BIND VARIABLES

DESCRIBE BIND VARIABLES describes the number of input host variables in an SQL statement and stores that number in an input SQLDA. You execute DESCRIBE BIND VARIABLES after you prepare the SQL statement and before you execute it or open the corresponding cursor. Use the following form of OPEN with DESCRIBE BIND VARIABLES:

```
EXEC SQL
    OPEN cursor_name USING DESCRIPTOR input_sqlda_pointer
```

Format

```
EXEC SQL
    DESCRIBE BIND VARIABLES FOR statement_name
    INTO input_sqlda_pointer
```

Argument	Description
statement_name	(required) Statement identifier of the prepared SQL statement whose variables are being described.
input_sqlda_pointer	(required) Pointer to the input SQLDA that will contain the number of input host variables in the select list.

Example

The following example describes the input host variables for a dynamic SELECT statement into an input SQLDA (isqldaptr is the pointer to this SQLDA):

```
EXEC SQL
    DESCRIBE BIND VARIABLES FOR dynstmt INTO isqldaptr ;
```

DESCRIBE SELECT LIST

DESCRIBE SELECT LIST describes the output variables in a SELECT statement and stores the information in an output SQLDA. You execute DESCRIBE SELECT LIST after you open the cursor for the associated SQL statement and before you issue a FETCH for the corresponding cursor.

DESCRIBE SELECT LIST stores the following information about output items in an SQLDA:

- Number of output items that are returned
- Data type of each output item
- Length of each output item
- Precision value for DECIMAL or NUMERIC output item
- Scale value for each DECIMAL or NUMERIC output item
- Null value indicator for each output item
- Name of each item in the select list

Format

```
EXEC SQL
    DESCRIBE SELECT LIST FOR statement_name
    INTO output_sqlda_pointer
```

Argument	Description
statement_name	(required) Statement identifier of the prepared SQL statement whose variables are being described.
output_sqlda_pointer	(required) Pointer to the output SQLDA that will contain the number of output host variables, their data types, and column lengths.

Example

The following example describes a select list for a dynamic SQL statement into an output SQLDA (osqldaptr is the pointer to this SQLDA):

```
EXEC SQL
    DESCRIBE SELECT LIST FOR dynstmt INTO osqldaptr ;
```

DISCONNECT

DISCONNECT terminates the connection between a program and a StorHouse database. You can terminate a specific connection, the current connection, or all established connections.

Format

```
EXEC SQL
    DISCONNECT {connection_name | ALL | CURRENT }
```

Argument	Description
connection_name	(required when ALL or CURRENT is omitted) Disconnects from a specific connection, expressed as a character literal or a host variable.
ALL	(required when connection_name or CURRENT is omitted) Disconnects from all established connections.
CURRENT	(required when connection_name or ALL is omitted) Disconnects from the current connection.

Examples

- The following example terminates the connection established by the conn_2 connect string:

  ```
  EXEC SQL
      DISCONNECT ‘conn_2’ ;
  ```

- The following example terminates all connections:

  ```
  EXEC SQL
      DISCONNECT ALL ;
  ```

DROP INDEX

DROP INDEX deletes an index from the specified user table. You can delete an index if you own the user table, use the SYSADM account, or have one of the following privileges:

- DBA
- RESOURCE
- INDEX privilege on the user table

Note the following:

- If the index is a value or hash index, this statement removes all index files (StorHouse primary files) for all corresponding segments.
- If the index is a range index, this statement removes all index entries for all corresponding segments from the range index system tables.
- You cannot use a ROLLBACK WORK statement to undo a DROP INDEX statement.

Format

DROP INDEX index_name [ON [owner.] table_name]

Argument	Description
index_name	(required) Name of the index that you want to remove.
owner.	(optional) Account ID of the owner of the table to which the index belongs. You must have DBA privilege to drop an index owned by another account.
table_name	(optional) Name of the table to which the index belongs.

Example

The following DROP INDEX statement removes the CUSTINDEX index on the CUSTOMER table from the current database.

```
DROP INDEX CUSTINDEX ON CUSTOMER
```

DROP SYNONYM

DROP SYNONYM removes the specified synonym. You must have DBA privilege or own the synonym to drop a private synonym. DBA privilege is required to drop a public synonym.

Note: You cannot use a ROLLBACK WORK statement to undo a DROP SYNONYM statement.

Format

DROP [PUBLIC] SYNONYM synonym

Argument	Description
PUBLIC	(optional) Indicates that you are dropping a public synonym. If you omit PUBLIC, StorHouse/RM assumes you want to remove a private synonym.
synonym	(required) Name of the synonym you want to remove.

Example

The following DROP SYNONYM statement removes a private synonym named CUSTOMER.

DROP SYNONYM CUSTOMER

DROP TABLE

DROP TABLE removes the specified user table from the current database. You must have DBA privilege or own the user table to use DROP TABLE. When you drop a user table, StorHouse/RM also drops any associated indexes and privileges and marks the StorHouse file for deletion. StorHouse/RM does not automatically drop any views dependent on the table, but they do become invalid.

Note: You cannot use a ROLLBACK WORK statement to undo a DROP TABLE statement.

Format

DROP TABLE [owner.]table_name

Argument	Description
owner.	(optional) Account ID of the owner of the table to be dropped. You must have DBA privilege to drop a table owned by another account.
table_name	(required) Name of the table to be removed.

Example

The following DROP TABLE statement removes the CUSTOMER user table from the current database.

DROP TABLE CUSTOMER

DROP TABLE SPACE

DROP TABLE SPACE removes an existing user tablespace from the current database. You must have DBA privilege to drop a user tablespace. Note the following:

- Before dropping a user tablespace, you must drop all user tables in that user tablespace. When you drop a user table, StorHouse/RM automatically drops the indexes associated with the user table.
- Dropping a user tablespace drops all subspaces.
- You cannot use a ROLLBACK WORK statement to undo a DROP TABLE SPACE statement.
- You cannot drop system and temporary tablespaces.
- TABLE SPACE can be one or two words.

Format

DROP TABLE SPACE tablespace_name

Argument	Description
tablespace_name	(required) Name of the user tablespace to be dropped. You must specify an existing tablespace name within the current database.

Example

The following DROP TABLE SPACE statement drops the BILL98 tablespace from the current database.

```
DROP TABLE SPACE BILL98
```

DROP VIEW

DROP VIEW removes the specified view from the current database. You must have DBA privilege or own the view to use DROP VIEW. When you drop a view, StorHouse/RM does not automatically drop dependent views; instead, it makes them invalid.

Note: You cannot use a ROLLBACK WORK statement to undo a DROP VIEW statement.

Format

DROP VIEW [owner.]view_name

Argument	Description
owner.	(optional) Account ID of the owner of the view. You must have DBA privilege to drop a view owned by another account.
view_name	(required) Name of the view to be removed.

Example

The following DROP VIEW statement removes a view named NEWCUSTOMERS from the current database.

```
DROP VIEW NEWCUSTOMERS
```

EXECUTE

EXECUTE executes a prepared (dynamic) non-SELECT SQL statement. To use EXECUTE, you must have the required privileges to execute the prepared SQL statement. Note that:

- You must always PREPARE and EXECUTE an SQL statement within the same transaction.
- You cannot specify output host variables on EXECUTE.
- You can execute a prepared SQL statement one or more times with different host variables by repeating calls to EXECUTE within the same transaction.

EXECUTE has two optional clauses:

- USING :host_variable [:indicator_variable] – Specifies input host variables and input indicator variables. Use this clause when your application knows the number of host and indicator variables and their data types at compile time. Specify host variable names in the same order as their associated host variable markers appear in the prepared SQL statement you are executing. See "PREPARE" on page 4-75 for more information about host variable markers.
- USING DESCRIPTOR input_sqlda_pointer – Specifies information about input host variables that are allocated at runtime. Use this clause when the number of input host variables and their data types are unknown until runtime.

Format

```
EXEC SQL
    EXECUTE statement_name
        [ USING { :host_variable [:indicator_variable]
        [, :host_variable [:indicator_variable] ]...
        | DESCRIPTOR input_sqlda_pointer } ]
```

Argument	Description
statement_name	(required) Statement identifier assigned by PREPARE. This identifier specifies the SQL statement to be executed.
USING	(required only when statement_name contains input host variables that are declared in your program's Declare Section) USING specifies:
:host_variable	(required with USING) Name of an input host variable in statement_name.
:indicator_variable	(optional) Name of the input indicator variable that is associated with the preceding input host variable.
USING DESCRIPTOR	(required only when statement_name contains input host variables that are allocated at runtime) USING DESCRIPTOR specifies:
input_sqlda_pointer	(required with USING DESCRIPTOR) Pointer to the SQLDA that contains information about the input host variables in statement_name.

Examples

- The following example deletes a row in the sysadm.syssmusers system table. The host variable in the USING clause is :accountid. The program declares this variable in the Declare Section. The associated host variable marker for :accountid is :account_id_marker, which appears in the SQL statement string sql_string.

```
EXEC SQL BEGIN DECLARE SECTION ;
    char sql_string [256] ;
    char account_id [12] ;
EXEC SQL END DECLARE SECTION ;
...

strcpy (sql_string, "DELETE FROM sysadm.syssmusers WHERE
            accountid = :account_id_marker") ;

EXEC SQL PREPARE  stmt1 FROM :sql_string ;

EXEC SQL EXECUTE  stmt1 USING :account_id ;
...
```

- The following example uses the USING DESCRIPTOR clause to identify the pointer to the SQLDA that contains information about input host variables:

```
EXEC SQL BEGIN DECLARE SECTION ;
    char stmt [256] ;
EXEC SQL END DECLARE SECTION ;
...
EXEC SQL PREPARE stmtid FROM :stmt ;
EXEC SQL EXECUTE stmt USING DESCRIPTOR sqldaptr ;
...
```

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE prepares and executes—in one step—an SQL statement represented as a statement string or host variable. It parses the specified statement string and reports any errors in the SQLCA. To use EXECUTE IMMEDIATE, you must have the required privileges to execute the associated SQL statement. Note the following:

- EXECUTE IMMEDIATE can only be embedded in an ESQL application program. It is an executable statement that cannot be prepared.
- If you execute the same SQL statement more than once, PREPARE and EXECUTE are more efficient than EXECUTE IMMEDIATE.

Format

```
EXEC SQL
    EXECUTE IMMEDIATE { :host_variable | statement_string }
```

Argument	Description
:host_variable	(required if statement_string is omitted) Name of the host variable that specifies the SQL statement to be prepared and executed.
statement_string	(required if :host_variable is omitted) Character string format of the SQL statement to be prepared and executed. The statement string must: ■ Be enclosed in quotes ■ Not contain host variable references or parameter markers ■ Not start with EXEC SQL ■ Not terminate with a semicolon

Example

The following example deletes a row in the syssmusers system table when the accountid column equals the character literal user1:

```
EXEC SQL
    EXECUTE IMMEDIATE
    “DELETE FROM sysadm.syssmusers WHERE accountid = ‘user1’” ;
```

FETCH

FETCH retrieves the result set (active set) for queries that return more than one row. It moves the cursor to the next row of the active set and retrieves the column values of the current row of the active set into specified host variables or the output SQLDA. You must open a cursor for a SELECT statement before you execute a FETCH for that cursor.

ESQL positions the cursor as follows:

- The first FETCH after an OPEN cursor positions the cursor at the first row in the result set. The first row then becomes the current row.
- Subsequent FETCH operations advance the cursor position one row in the active set, making that row the current row.
- You can move the cursor forward in the active set only by executing subsequent FETCH statements.
- You can move the cursor to the beginning of the active set only by closing and then reopening the cursor.

FETCH has two formats: INTO and USING DESCRIPTOR. Use the INTO clause if the statement contains known output host variables. Use the USING DESCRIPTOR clause if you are using an SQLDA.

If the cursor is positioned on the last row of the active set or if the active set does not contain any rows, FETCH returns status code 100 in the SQLCA. This indicates that there are no more rows to be fetched or the SQL_NOT_FOUND condition. Upon successful execution, FETCH returns the cumulative number of rows fetched so far for a given cursor since the last open in the SQLCA field sqlerrd[8].

You can fetch multiple rows in one FETCH call by using ESQL array variables in the INTO clause. In this case, FETCH returns status code 100 in the SQLCA when it reaches the end of the active set, even if the current execution of FETCH

returns one or more rows. When you use ESQL array variables with FETCH, ESQL automatically sets the argument tpe_size in the C structure that represents the array to the number of rows that were actually fetched. If you use C instead of SQL to fetch, you must manually set tpe_size to the number of rows that were fetched.

Format

```
EXEC SQL
    FETCH cursor_name
    { INTO :host_variable [:indicator_variable]
    [,:host_variable [:indicator_variable] ]...
    | USING DESCRIPTOR output_sqlda_pointer }
```

Argument	Description
cursor_name	(required) Name of the cursor associated with the prepared SELECT statement.
:host_variable	(required with INTO) Name of the output host variable that will contain a column value from the active set.
:indicator_variable	(optional) Name of the output indicator variable associated with the preceding host variable.
output_sqlda_pointer	(required with USING DESCRIPTOR) Pointer to the SQLDA that defines storage areas for the output host variables.

Examples

- The following example fetches the current row in the active set into the output host variables cust_no_v, name_v, street_v, city_v, and state_v:

```
EXEC SQL BEGIN DECLARE SECTION ;
    long cust_no_v ;
    char name_v [20] ;
    char street_v [40] ;
    char city_v [10] ;
    char state_v [2] ;
EXEC SQL END DECLARE SECTION ;
...
EXEC SQL OPEN cust_cur ;
for (;;)
{
    EXEC SQL
        FETCH cust_cur
        INTO :cust_no_v, :name_v, :street_v, :city_v, :state_v ;
...
}
...
```

- The following example uses the output SQLDA (sqldaptr is the pointer to this SQLDA) to obtain information about the output host variables:

```
...
EXEC SQL
    FETCH cust_cur USING DESCRIPTOR sqldaptr ;
...
```

- The following example declares a locator variable named hv_prod_locator for CLOB data:

```
EXEC SQL BEGIN DECLARE SECTION ;
    CLOB_LOCATOR hv_prod_locator ;
    CLOB(2M)hv_product ;
EXEC SQL END DECLARE SECTION ;
```

Then declares and opens a cursor called my_cursor to execute the query:

```
EXEC SQL
DECLARE my_cursor CURSOR FOR
    SELECT ProdDescr
    FROM ProdInfo
    WHERE ProdName LIKE “%Wrench%” ;
EXEC SQL
    OPEN my_cursor ;
```

And finally associates the result with the locator variable:

```
EXEC SQL
    FETCH my_cursor
    INTO :hv_prod_locator ;
```

FREE LOCATOR

FREE LOCATOR releases one or more locator variables before the end of a transaction, freeing the server storage used by the locator variable. If you do not explicitly release a locator variable, StorHouse/RM releases it at the end of the transaction. You can continue to use a locator variable as long as it has not been released with FREE LOCATOR or the transaction has not ended.

Format

```
EXEC SQL
    FREE LOCATOR :locator_variable [, :locator_variable]...
```

Argument	Description
:locator_variable	(required) Name(s) of one or more locator variables to be released.

Example

The following example releases the LOB locators called :product_locator and :product_desc_locator.

```
EXEC SQL
    FREE LOCATOR :product_locator, :product_desc_locator ;
```

GRANT

GRANT assigns database or database component privileges to the specified StorHouse accounts.

Note: You cannot use a ROLLBACK WORK statement to undo a GRANT statement.

Format

The GRANT statement has two formats. The first format grants database privileges to specific accounts. You must have DBA privilege to use this format.

```
GRANT {RESOURCE, DBA, SCAN}
TO account_id [,account_id]...
```

The second format grants privileges to accounts for tables and views in the database. You must own the tables and views or have DBA privilege to use this format.

```
GRANT {privilege [,privilege]... | ALL}
ON {table_name | view_name}
TO {account_id [,account_id]...  | PUBLIC}
[WITH GRANT OPTION]
```

where privilege is defined as:

```
{DELETE | INDEX | INSERT | SELECT
| UPDATE [(column [,column]... )]}
```

Argument	Description
RESOURCE	(required if you omit DBA and SCAN) Grants RESOURCE privilege, which enables the specified account(s) to create indexes (on tables the account owns), user tables, and views.
DBA	(required if you omit RESOURCE and SCAN) Grants DBA privilege, which enables the specified account(s) to access and modify any table or view in the database. DBA privilege includes all the privileges provided by the RESOURCE privilege but does not include SCAN privilege.
SCAN	(required if you omit RESOURCE and DBA) Grants SCAN privilege, which enables the specified account(s) to execute queries that scan all rows of a user table (full table scans).
privilege	(required if ALL is omitted) Type of privilege(s) you are granting. Valid values are:
DELETE	Grants DELETE privilege, which enables the specified account(s) to use DELETE on the specified system table or system table view.
INDEX	Grants INDEX privilege, which enables the specified account(s) to create an index on the specified table.
INSERT	Grants INSERT privilege, which enables the specified account(s) to insert rows in the specified system table or load data into the specified user table/view.
SELECT	Grants SELECT privilege, which enables the specified account(s) to perform queries on the specified table/view.
UPDATE	Grants UPDATE privilege, which enables the specified account(s) to use UPDATE on the specified system table or system table view. You may limit this privilege to specified columns of the system table/view.
column	(required) Name of the column for which UPDATE privilege is granted.
ALL	(required if an individual privilege is omitted) Grants DELETE, INDEX, INSERT, SELECT, and UPDATE privileges.
table_name	(required if view_name is not specified) Name of the table on which you are granting the privilege(s).
view_name	(required if table_name is not specified) Name of the view on which you are granting the privilege(s).

Argument	Description
account_id	(required if PUBLIC is omitted) Account ID of the user to receive the new privilege(s).
PUBLIC	(required if account_id is omitted) Grants the specified privileges on the table to every valid StorHouse account.
WITH GRANT OPTION	(optional) Allows the specified account(s) to grant their access rights or a subset of their rights to other accounts.

Examples

- The following GRANT statement grants DBA privilege to two accounts: USER1 and USER2.

  ```
  GRANT DBA
  TO USER1, USER2
  ```

- The following GRANT statement grants INDEX privilege on a table named CUST_VIEW to an account named DBUSER1.

  ```
  GRANT INDEX
  ON CUST_VIEW
  TO DBUSER1
  ```

INSERT

INSERT inserts new rows into the specified system table or system table view. You cannot insert rows into user tables. You specify the values for the inserted rows. You must have DBA privilege, own the system table, or have INSERT privilege on the system table to use INSERT. To specify a query expression, you must have DBA privilege or have SELECT privilege on all the tables/views referred to in the query expression.

Format

```
INSERT INTO [owner.]{table_name | view_name}
[(column_name [,column_name]...)]
{VALUES (value [,value]...) |
query_expression}
```

Argument	Description
owner.	(optional) Account ID of the owner of the system table or system table view. If you are not the owner, you must have DBA privilege or INSERT privilege to insert new rows.
table_name	(required for system table inserts) Name of the system table.
view_name	(required for system view inserts) Name of the system table view.

Argument	Description
column_name	(optional or required depending on other specifications) Name of the column to receive the inserted values. ■ If you specify one or more column names, you must supply values for those columns only. In this case, value specification must match the order of column specification. StorHouse/RM places NULL values in the other columns of the inserted row, provided the column definition allows NULL values and no default definitions exist for the columns. ■ If a default definition exists for a column and the column is not included in the optional list, StorHouse/RM places the default value in the column. ■ If you omit column name, you must specify values for all columns. In this case, value specification order must match the order of column specification when the table was created.
value	(required) Actual values you want to insert. You can insert only one row of specified values at a time.
query_expression	(optional) SELECT statement that selects values from another table or view.

Example

The following INSERT statement inserts a new account ID called DBUSER1 and a default user tablespace called USER1TS in the SYSSMUSERS system table.

```
INSERT INTO SYSADM.SYSSMUSERS (ACCOUNTID, DEFAULT_TS)
VALUES ('DBUSER1', 'USER1TS')
```

OPEN

OPEN executes the query that is associated with the specified cursor and identifies the rows in the result set (also called the active set). OPEN places the cursor in an open state just before the first row of the active set. Once the cursor is opened, the active set does not change, and host variables are not re-examined. If you subsequently change a host variable value and need to determine a new active set, you must close and reopen the cursor. OPEN requires DBA privilege or SELECT privilege on all tables/views specified in the associated SELECT statement.

OPEN has an optional USING clause for specifying host variables. The USING clause has two formats:

- USING :host_variable [:indicator_variable] – Provides information about host variables in the prepared query. Specify host variable names in the USING clause in exactly the same order as their associated host variable markers are specified in the SQL statement associated with the same cursor. See "PREPARE" on page 4-75 for more information about host variable markers.

- USING DESCRIPTOR input_sqlda_pointer – Provides information about host variables that are allocated at runtime.

Note the following:

- You cannot open a cursor that is already in the open state.

- DECLARE and OPEN for the same cursor must occur within the same transaction.

- If the DECLARE statement contains references to host variables, the associated OPEN statement must occur within the same scope as that of the referenced variables.

Format

```
EXEC SQL
    OPEN cursor_name
        [ { USING :host_variable [:indicator_variable]
        [,:host_variable [:indicator_variable] ]...
        | USING DESCRIPTOR input_sqlda_pointer } ]
```

Argument	Description
cursor_name	(required) Name of the cursor to be opened.
USING	(required only if the prepared SQL statement associated with cursor_name contains input host variables that are declared in your program's Declare Section) USING specifies:
:host_variable	(required with USING) Name of an input host variable in the prepared SQL statement associated with cursor_name.
:indicator_variable	(optional) Name of the input indicator variable that is associated with the preceding input host variable.
USING DESCRIPTOR	(required only if the prepared SQL statement associated with cursor_name contains input host variables that are allocated at runtime) USING DESCRIPTOR specifies:
input_sqlda_pointer	(required with USING DESCRIPTOR) A pointer to the SQLDA that contains information about the input host variables in the prepared SQL statement associated with cursor_name.

Examples

- The following example opens the cursor cust_cursor:

```
EXEC SQL
    OPEN cust_cur ;
```

- The following example opens the cursor ord_cur and specifies the input host variable order_no_v. This input host variable must have been declared in your program's Declare Section.

```
EXEC SQL
    OPEN ord_cur USING :order_no_v ;
```

- The following example opens the cursor dyn_cur and specifies a pointer (sqldaptr) to the input SQLDA that contains information about input host variables.

```
EXEC SQL
    OPEN dyn_cur USING DESCRIPTOR sqldaptr ;
```

PREPARE

PREPARE parses an SQL statement for syntax errors and then assigns an identifier to the statement. The parser returns an error in the SQLCA if the statement syntax is incorrect. You prepare an SQL statement once and then execute it as often as necessary within the same transaction. If the current transaction is committed or rolled back and the SQL statement is to be re-executed, you must prepare the statement again. PREPARE requires DBA privilege or the privilege to execute the statement being prepared.

Note: If the prepared statement is a SELECT, use OPEN, FETCH and CLOSE instead of EXECUTE to obtain the result set.

Format

```
EXEC SQL
    PREPARE statement_name FROM string_variable
```

where string_variable is defined as:

```
character_string | :host_variable
```

Argument	Description
statement_name	(required) Assigned identifier for the SQL statement represented by string_variable. Statement_name must be unique within a file.
string_variable	(required) String variable (character string or host variable) that contains the SQL statement being prepared. If you specify a host variable, you must declare it as a character array in a Declare Section.

Examples

- The following example prepares an SQL statement from a character string:

```
EXEC SQL
    PREPARE delstmt FROM ‘delete from sysadm.syssmusers
    where accountid=:mkr1’ ;
```

- The following example prepares and executes the SQL statement represented by the string variable sql_string. The substitution marker :id_marker is a place holder for the host variable :account_id, which appears in the USING clause.

```
...
EXEC SQL BEGIN DECLARE SECTION ;
    char sql_string [256] ;
    char account_id [12] ;
EXEC SQL END DECLARE SECTION ;

strcpy ( sql_string,
    "DELETE FROM sysadm.syssmusers WHERE accountid =
:id_marker") ;

EXEC SQL PREPARE  del_acct_stmt FROM :sql_string ;
EXEC SQL EXECUTE  del_acct_stmt USING :account_id ;

...
```

REVOKE

REVOKE removes the specified access privileges from the specified StorHouse accounts in a database. If one account was granted access to a table by several other accounts, all those other accounts must issue a REVOKE for the account to lose access to the table. Revoking an account's access privileges also revokes any privileges that account may have given to others.

Note: You cannot use a ROLLBACK WORK statement to undo a REVOKE statement.

Format

The REVOKE statement has two formats. The first format revokes database privileges from specified accounts. You must have DBA privilege to use this format.

```
REVOKE {RESOURCE, DBA, SCAN}
FROM account_id [,account_id]...
```

The second format revokes various access privileges to tables and views in a database. To use this format, you must own the tables and views, have DBA privilege, or have been granted the privileges by another account through the WITH GRANT OPTION.

```
REVOKE {privilege [,privilege]... | ALL}
ON {table_name | view_name}
FROM {account_id [,account_id]... | PUBLIC}
```

where privilege is defined as:

```
{DELETE | INDEX | INSERT | SELECT
| UPDATE [(column [,column]... )]}
```

Argument	Description
RESOURCE	(required if you omit DBA and SCAN) Revokes RESOURCE privilege.
DBA	(required if you omit RESOURCE and SCAN) Revokes DBA privilege.
SCAN	(required if you omit RESOURCE and DBA) Revokes SCAN privilege.
account_id	(required if PUBLIC is omitted) Account ID of the user whose privilege(s) is being revoked.
privilege	(required unless you specify ALL) Type of privilege(s) being revoked. Valid values are:
DELETE	Revokes DELETE privilege.
INDEX	Revokes INDEX privilege.
INSERT	Revokes INSERT privilege.
SELECT	Revokes SELECT privilege.
UPDATE	Revokes UPDATE privilege.
column	Name of the column for which UPDATE privilege is revoked.
ALL	(required unless you specify at least one individual privilege) Revokes DELETE, INDEX, INSERT, SELECT, and UPDATE privileges.
table_name	(required if view_name not specified) Name of the table from which you are revoking privileges.
view_name	(required if table_name not specified) Name of the view from which you are revoking privileges.
PUBLIC	(required if an account_id is omitted) Revokes the specified rights on the table or view from every StorHouse account.

Examples

- The following REVOKE statement revokes SCAN privilege from an account named DBUSER1.

  ```
  REVOKE SCAN
  FROM DBUSER1
  ```

- The following REVOKE statement revokes INDEX privilege on a table named CUST_VIEW from an account named DBUSER2.

  ```
  REVOKE INDEX
  ON CUST_VIEW
  FROM DBUSER2
  ```

ROLLBACK WORK

ROLLBACK WORK cancels the current transaction and rolls back any database changes performed during the transaction. Note the following:

- You cannot use ROLLBACK WORK after issuing the following SQL statements:
  - ALTER TABLE SPACE
  - CREATE INDEX
  - CREATE SYNONYM
  - CREATE TABLE
  - CREATE TABLE SPACE
  - CREATE VIEW
  - DROP INDEX
  - DROP SYNONYM
  - DROP TABLE
  - DROP TABLE SPACE
  - DROP VIEW
  - GRANT
  - REVOKE
- All locks held by the transaction are released when the transaction is rolled back.
- A transaction is automatically rolled back in the event of a hardware failure, software failure, or lock time-out error.
- If an active transaction exists when an ESQL application disconnects from a database, StorHouse/RM automatically rolls back the transaction.
- A rollback operation closes all cursors and releases (frees) all LOB locators opened during the transaction.

Format

```
EXEC SQL
    ROLLBACK WORK
```

Example

The following example cancels the change to the sysadm.syssmusers system table:

```
EXEC SQL
    UPDATE sysadm.syssmusers
    SET default_ts = :def_tbspace
    WHERE accountid = :acct_id ;

EXEC SQL
    ROLLBACK WORK ;
```

SELECT

SELECT retrieves information from one or more tables in a database. The column names and expressions that follow the SELECT keyword make up your *select list*. You can perform a join (see page 4-94) when you specify multiple table names. You can use set operators to combine two SELECT statements for more complex queries.

You must have DBA privilege, own the table, or have SELECT privilege on the table to use SELECT. You cannot execute a query that requires a full table scan unless you have SELECT privilege on the user table and the database-level SCAN privilege. SCAN privilege is often provided only on an as needed basis because full table scans are resource-intensive and can affect system performance.

Note the following:

- Certain types of queries qualify for extractor processing. Refer to the *StorHouse ESQL Manual* for more information about the StorHouse extractor.
- LOB columns cannot be used to join two tables, and LOB data types are not allowed with the following SELECT statement clauses: DISTINCT, ORDER BY and GROUP BY.

Format

```
SELECT [ALL | DISTINCT]
{* | expr [column_alias] [, expr [column_alias] ]...}
[INTO :host_variable [,:host_variable]...] ]
[FROM table_spec [, table_spec ]...]
[WHERE condition]
[GROUP BY column_name [,column_name]... [HAVING condition] ]
[ {UNION | UNION ALL } SELECT...]
[ORDER BY {expr | position} [ASC | DESC] [, {expr | position} [ASC | DESC]
]... ]
[FOR {FETCH | READ} ONLY]
```

Argument	Description
ALL	(optional) Returns all selected rows, including duplicates. If you omit ALL and DISTINCT, the default is ALL.
DISTINCT	(optional) Returns only unique rows. LOB data types are not allowed with DISTINCT.
*	(required if table.column_alias and expr are omitted) Indicates to select from all columns in the specified table(s).
expr	(required if * and column_alias are omitted) Item to be selected presented as an identifier, function, value, or other valid type of expression.
column_alias	(required if * and expr are omitted) Name of the column to be selected by the query.
INTO :host variable	(required for an embedded SQL query that returns one row) Clause indicating the name(s) of the host variables that receive the selected data. The INTO clause, if present, must precede the FROM clause.
FROM	(optional) Clause indicating the table(s) to be accessed and any join information. If you omit this clause, the select list can contain only constants and special registers.
WHERE	(optional) Clause indicating the search conditions (predicates) to be used for row selection. If you omit this clause, the query returns all rows of the specified table (if the user has SCAN privilege on the user table) or the Cartesian product of all tables specified in the FROM clause.
GROUP BY column_name	(optional) Clause that specifies how to group rows returned by the query. LOB data types are not allowed with GROUP BY.
HAVING	(optional) Clause that applies one or more qualifying conditions for selected groups.

Argument	Description
set_operator	(optional) Specifies the action to apply on two sets of rows that are returned by separate queries. The set_operator argument is preceded and followed by a SELECT statement. You cannot use set operators in subqueries. Set operators are: ■ UNION – Unites the output of two or more queries into a single set of rows and columns, excluding duplicate rows from the output. ■ UNION ALL – Unites the output of two or more queries into a single set of rows and columns, including duplicate rows in the output.
ORDER BY	(optional) Clause that specifies the order of the rows selected by the query. Include ORDER BY to ensure a specific row order in a result set. LOB data types are not allowed with ORDER BY.
FOR	(optional) Clause allowing IBM DB2 programs to work correctly without changing the SQL.

INTO clause

INTO specifies the names of the output host variables to receive the data retrieved by a static SELECT statement. You can use the INTO clause in a SELECT statement that returns one row only. If the SELECT statement returns multiple rows, you must include an INTO clause as part of the FETCH statement. The INTO clause must precede the FROM clause.

Format

INTO :host_variable [,:host_variable]...

Argument	Description
:host_variable	(required) Name of the output host variable to contain the resulting value. The number of output host variables must equal the number of specified columns.

Examples

- The following SELECT statement selects a row from the customer table where the cust_no column contains the specified number. The resulting cust_name value is assigned to output host variable :cust_v.

```
SELECT cust_name
    INTO :cust_v
    FROM customer
    WHERE cust_no=8973205;
```

- The following SELECT statement associates the result (a product description) with the locator variable :hv_prod_locator.

```
SELECT ProdDescr
    INTO :hv_prod_locator
    FROM ProdInfo
    WHERE ProdName LIKE “%Wrench%” ;
```

FROM clause

FROM specifies the table(s) or view(s) to be accessed as well as any join specifications. See “Joins” on page 4-94 for more information about joining tables.

Format

FROM table_spec [, table_spec]...

where table_spec is:

table_reference [correlation] | joined_table

Argument	Description
table_reference	(required) Name of the table.
correlation	(optional) Correlation name for a table participating in a join. The format is: [AS] correlation_name
joined_table	(optional) Join specification. The format is: (joined_table) | table_spec { [INNER] | LEFT [OUTER] } JOIN table_spec ON join_condition
(joined_table)	Anything that qualifies as a joined_table can be enclosed in parentheses and considered as a joined_table itself. For example: FROM pilot JOIN (service JOIN plane ON plane.serial_num = service.serial_num) ON plane.serial_num = pilot.serial_num
INNER JOIN	Join operator that specifies an inner-join operation. When the join condition is true, the matched rows of the tables are combined. The unmatched rows are omitted from the result table. The following operators are valid for inner-join: ■ JOIN ■ INNER JOIN

Argument	Description
LEFT OUTER JOIN	Join operator that specifies a left outer-join operation. When the join condition is true, the matched rows of the tables are combined (like an inner-join) and the unmatched rows of the table to the left of the join operator are preserved, combined with NULL values for the columns in the table to the right of the join operator. The following operators are valid for outer-join: ■ LEFT JOIN ■ LEFT OUTER JOIN
join_condition	Search condition, or predicate, that evaluates to true, false, or unknown for a given row. You can specify multiple predicates with logical operators AND and OR. The join condition cannot contain a subquery.

Example

The following SELECT statement selects all rows from the AGENTS table where the COMMISSION earned is between 10% and 12% inclusive.

```
SELECT *
FROM AGENTS
WHERE COMMISSION BETWEEN .10 AND .12
```

WHERE clause

WHERE specifies the restrictions to be applied for row selection.

Format

WHERE condition

Argument	Description
condition	(required) Condition or predicate that evaluates to true or false for a given row or group. StorHouse/RM applies the condition to each row of the result set of the WHERE clause and selects only those rows that satisfy the condition for the result set. See Chapter 5, “StorHouse SQL predicates,” for more information about predicates.

Example

The following SELECT statement selects all rows from the CUSTOMER table where the CITY is BURLINGTON and STATE is MA.

```
SELECT *
FROM CUSTOMER
WHERE CITY='BURLINGTON' AND STATE='MA'
```

GROUP BY clause

GROUP BY combines groups of rows into summary results. It allows you to define a subset of values in one column in terms of another column, and then apply an aggregate function to that subset. When you use GROUP BY, the SELECT statement select list can contain only those columns specified in GROUP BY, aggregate functions on any column, or both.

Format

GROUP BY column_name [,column_name]...

Argument	Description
column_name	(required) Name of the column(s) used to group the results. The column can be any data type except BLOB or CLOB.

Example

The following SELECT statement finds each subscriber's largest telephone bill.

```
SELECT SUBSCRIBER MAX(AMT)
FROM BILLING
GROUP BY SUBSCRIBER
```

HAVING clause

HAVING applies one or more qualifying conditions to groups specified in the GROUP BY clause. (HAVING is like a WHERE clause for a GROUP BY clause.) If HAVING is used without GROUP BY, the implicit group refers to all rows returned by the SELECT statement WHERE clause.

Format

HAVING condition

Argument	Description
condition	(required) Compares one aggregate function value with another aggregate function value or a literal. Arguments specified in a condition must have a single value per output group. Refer to Chapter 6, “StorHouse SQL functions,” for more information about aggregate functions.

Example

The following SELECT statement finds the maximum sales over 5000.00 for each customer sales representative on each order date.

```
SELECT CUST_REP, ODATE, MAX(SALES)
FROM ORDERS
GROUP BY CUST_REP,ODATE
HAVING MAX(SALES)>5000.00
```

ORDER BY clause

ORDER BY specifies how to order the rows in a result set. Including ORDER BY is the only way to ensure the row order of a result set. If you identify only one column, the rows are ordered by the values of that column. If you identify multiple columns, the rows are ordered by the values of the first specified column, then by the values of the second, and so on. A NULL value is considered lower than all other values.

Note: If a query contains set operators, you must specify only a column position in the ORDER BY clause, rather than a column name.

Format

[ORDER BY {expr | position} [ASC | DESC] [,{expr | position}[ASC | DESC]]...

Argument	Description
expr	(required) Item used to order rows in the result set, presented as an identifier, function, value, or other valid type of expression. The expression can be any data type except BLOB or CLOB.
position	(required if expr is omitted) Integer that represents the placement of a column or expression in the select list.
ASC	(optional) Specifies an ascending order for the returned rows. If ASC and DESC are omitted, the default is ASC.
DESC	(optional) Specifies a descending order for the returned rows.

Examples

- The following SELECT statement selects name and address information from the CUSTOMER table and returns the results in ascending order according to NAME.

  ```
  SELECT NAME, STREET, CITY, STATE, ZIP
  FROM CUSTOMER
  ORDER BY NAME
  ```

- The following SELECT statement selects first names, last names, and completed sales from the SALES_EAST and SALES_WEST tables. The statement returns the results in ascending order by LAST_NAME.

  ```
  SELECT FIRST_NAME, LAST_NAME, COMPLETED_SALES
  FROM SALES_EAST
  UNION
  SELECT FIRST_NAME, LAST_NAME, COMPLETED_SALES
  FROM SALES_WEST
  ORDER BY 2
  ```

FOR clause

StorHouse SQL supports the FOR clause in the SELECT statement syntax only for compatibility with IBM DB2 SQL. DB2 programs that use a SELECT statement with the FOR clause to access StorHouse database tables will be parsed correctly. The FOR clause has no effect on SELECT statement execution, even when a system table is included in the SELECT statement table list. The FOR clause must follow the FROM, WHERE, GROUP BY, HAVING, and ORDER BY clauses and any set operators in the SELECT statement syntax.

Format

[FOR {FETCH | READ} ONLY]

Joins

A *join* is a SELECT statement that combines data in multiple tables or views or within a table or view. StorHouse/RM supports the ANSI SQL join syntax. You write queries with explicit join operations in the FROM clause of a SELECT statement, using an ON clause to specify the join condition. StorHouse/RM currently supports inner-join and left outer-join operations.

Basic guidelines for writing queries are as follows:

- You can perform multiple join operations in the same query, for instance, multiple inner-join operations, multiple outer-join operations, or a combination of the two.
- Any column referenced in a join condition must be a column in one of the tables of the associated join operation.
- Table order is significant for outer-join operations, insignificant for inner-join operations.
- For left outer-joins, the table referenced to the left of the join operator is the preserved table.
- You can use parentheses to specify the sequence to perform join operations.
- The WHERE clause has a different effect on query results from the ON clause.

The next several sections contain examples that illustrate these basic guidelines.

Sample tables

The join examples in this document reference the following tables.

Pilot table

firstname	lastname	serial_num
John	Smith	
Mary	Jones	DC1001
Brad	Knickerbocker	DC1002
Lucy	Hayes	BOE001
Rutherford	Hayes	BOE002

Plane table

serial_num	descr
DC1001	DC 10
DC1002	DC 10
BOE001	Boeing 747
BOE002	Boeing 737
FOK001	Fokker

Service table

serial_num	descr
DC1002	Oil Change
FOK001	New Propeller

Performing multiple inner-join operations

StorHouse/RM supports queries with multiple join operations. Joins are processed in order based on the position of the join condition (ON clause). You can also use parentheses (see the example on page 4-99) to specify join order.

For example, the following query, which joins the Service, Plane, and Pilot tables, locates the names of the pilots to be notified that their planes require service.

```
SELECT firstname, lastname, plane.serial_num
FROM service INNER JOIN plane ON service.serial_num = plane.serial_num
INNER JOIN pilot ON plane.serial_num = pilot.serial_num
```

This query consists of two inner-join operations. The first inner-join operation joins the Service table and the Plane table on the serial_num column. The ON clause for this inner-join operation may only reference the columns in the participating tables. In other words, referencing a column in the Pilot table would be invalid because the Pilot table is outside the scope of this ON clause. The result of the first inner-join is this intermediate table:

Intermediate table joining Service and Plane

serial_num	descr	serial_num	descr
DC1002	Oil Change	DC1002	DC 10
FOK001	New Propeller	FOK001	Fokker

The second inner-join operation joins the intermediate table with the Pilot table. The scope of the second ON clause includes references to any columns in the Service table and the Plane table on the one side of the INNER JOIN operator and any columns in the Pilot table on the other side of the join operator. The result of the second inner-join operation is this result table:

Result of inner-join operations

firstname	lastname	serial_num
Brad	Knickerbocker	DC1002

Performing a left outer-join

A left outer-join operation includes all matched rows as well as unmatched rows, preserving the unmatched rows in the left table (the table before the join operator) by adding NULL values for the right table columns (the table after the join operator). A left outer-join means that the preserved table appears to the left of the join operator.

For example, the following query determines pilots and their aircraft assignments as well as pilots without aircraft assignments.

```
SELECT firstname, lastname, plane.serial_numserial_num, descr
FROM pilot LEFT OUTER JOIN plane
ON pilot.serial_num = plane.serial_num
```

In this example, the result table contains all rows from the Pilot table and NULL values for unmatched rows in the Plane table. All pilots, except John Smith, have plane assignments.

Result of left outer-join

firstname	lastname	serial_num	descr
John	Smith		
Mary	Jones	DC1001	DC 10
Brad	Knickerbocker	DC1002	DC 10
Lucy	Hayes	BOE001	Boeing 747
Rutherford	Hayes	BOE002	Boeing 737

Changing the table order

For an inner-join operation, the order of the table references in the FROM clause is insignificant. You can specify the tables in any order and the final result set is the same.

For example, both of the following inner-join operations produce the same result:

```
SELECT firstname, lastname, plane.serial_num
FROM service INNER JOIN plane ON service.serial_num = plane.serial_num
INNER JOIN pilot ON plane.serial_num = pilot.serial_num
```

```
SELECT firstname, lastname, plane.serial_num
FROM pilot INNER JOIN service ON pilot.serial_num = service.serial_num
INNER JOIN plane ON plane.serial_num = pilot.serial_num
```

Result of both inner-join operations

firstname	lastname	serial_num
Brad	Knickerbocker	DC1002

For an outer-join, however, the order in which tables are joined and the order in which they appear in the FROM clause is significant. The results differ depending on the order in which the tables are joined. For example, the following outer-join operation produces a result table with two rows:

```
SELECT firstname, lastname, plane.serial_num, service.descr
FROM service LEFT JOIN plane ON service.serial_num = plane.serial_num
LEFT JOIN pilot ON pilot.serial_num = plane.serial_num
```

Result of outer-join on Service, Plane, and Pilot

firstname	lastname	serial_num	descr
Mary	Jones	DC1001	Oil Change
		FOK001	New Propeller

Changing the table order in the FROM clause produces a different result:

```
SELECT firstname, lastname, plane.serial_num, service.descr
FROM pilot LEFT JOIN plane ON pilot.serial_num = plane.serial_num
LEFT JOIN service ON service.serial_num = pilot.serial_num
```

Result of outer-join on Pilot, Plane, and Service

firstname	lastname	serial_num	descr
John	Smith		
Mary	Jones	DC1001	Oil Change
Brad	Knickerbocker	DC1002	
Lucy	Hayes	BOE001	
Rutherford	Hayes	BOE002	

Using parentheses to specify join order

You can use parentheses to specify precedence or to improve the readability of the join operations. For example, the inner-join operation within parentheses is processed before the one outside of the parentheses:

```
SELECT firstname, lastname, plane.serial_num
FROM pilot JOIN
(service JOIN plane ON plane.serial_num = service.serial_num)
ON plane.serial_num = pilot.serial_num
```

Another way to think of this in this example is:

```
pilot JOIN intermediate_table
```

where intermediate_table is the result of the inner-join on the Service and Plane tables. The ON clause within the parentheses applies to the inner-join of Service and Plane. Only columns from these two tables are in the scope of the join condition for this ON clause. The scope of the ON clause outside the parentheses

may include references to columns in the Pilot table and columns in the tables within the parentheses.

Combining inner-join and outer-join operations

You can combine inner-join with outer-join operations in the same query. For example:

```
SELECT firstname, lastname, plane.serial_num, service.descr
FROM pilot INNER JOIN plane ON pilot.serial_num = plane.serial_num
LEFT JOIN service ON service.serial_num = pilot.serial_num
```

Here's the intermediate result from the inner-join on Pilot and Plane:

Intermediate table of inner-join on Pilot and Plane

firstname	lastname	serial_num
Mary	Jones	DC1001
Brad	Knickerbocker	DC1002
Lucy	Hayes	BOE001
Rutherford	Hayes	BOE002

Here's the result of the left outer-join with the intermediate table and Service.

Result of the join

firstname	lastname	serial_num	descr
Mary	Jones	DC1001	Oil Change
Brad	Knickerbocker	DC1002	
Lucy	Hayes	BOE001	
Rutherford	Hayes	BOE002	

Using a WHERE clause with a join

A WHERE clause, which is optional and additional to the ON clause, affects the final result set. The ON clause specifies the join condition. The WHERE clause further restricts the result of the final joined table. The ON clause must reference only the tables in the join condition. The WHERE clause can reference any table in the query.

For example, the following outer-join operations produce different results. The first query contains an ON clause with two predicates, no WHERE clause:

```
SELECT * FROM pilot LEFT OUTER JOIN plane
ON pilot.serial_num = plane.serial_num AND plane.descr='DC 10'
```

The second query contains an ON clause with one predicate and a WHERE clause:

```
SELECT * FROM pilot LEFT OUTER JOIN plane
ON pilot.serial_num = plane.serial_num
WHERE plane.descr='DC 10'
```

Explanation of query 1. The first step in an outer-join operation is to combine the matching rows, that is, to perform an inner-join. Then the preserved rows are added to the final result table. In the first query, the join condition (ON clause) consists of two predicates.

```
SELECT * FROM pilot LEFT OUTER JOIN plane
ON pilot.serial_num = plane.serial_num AND plane.descr='DC 10'
```

Only two rows in both tables match the join condition (serial_nums match and descr is DC 10).

Intermediate table of inner-join between Pilot and Plane

firstname	lastname	serial_num	serial_num	descr
Mary	Jones	DC1001	DC1001	DC 10
Brad	Knickerbocker	DC1002	DC1002	DC 10

For an outer-join, the table to the left of the join operator is the preserved table. In this example, the Pilot table is the preserved table. These three unmatched rows in the Pilot table will be preserved in the final result table:

Unmatched rows in the Pilot table

firstname	lastname	serial_num
John	Smith	
Lucy	Hayes	BOE001
Rutherford	Hayes	BOE002

The result table is a union of the intermediate table and the unmatched rows in the Pilot table along with NULL values for the columns in the Plane table.

Result table of query 1

firstname	lastname	serial_num	serial_num	descr
Lucy	Hayes	BOE001		
Rutherford	Hayes	BOE002		
Mary	Jones	DC1001	DC1001	DC 10
Brad	Knickerbocker	DC1002	DC1002	DC 10
John	Smith			

Explanation of query 2. In the second query, the join condition (ON clause) consists of one predicate.

```
SELECT * FROM pilot LEFT OUTER JOIN plane
ON pilot.serial_num = plane.serial_num
WHERE plane.descr='DC 10'
```

Four rows in both tables match the join condition.

Intermediate table of the inner-join between Pilot and Plane

firstname	lastname	serial_num	serial_num	descr
Lucy	Hayes	BOE001	BOE001	Boeing 747
Rutherford	Hayes	BOE002	BOE002	Boeing 737
Mary	Jones	DC1001	DC1001	DC 10
Brad	Knickerbocker	DC1002	DC1002	DC 10

Only one row in the Pilot table is unmatched:

Unmatched row in the Pilot table

firstname	lastname	serial_num
John	Smith	

The union of the preserved row in the Pilot table with the intermediate table produces the following:

Outer-join of Pilot and intermediate table

firstname	lastname	serial_num	serial_num	descr
Lucy	Hayes	BOE001	BOE001	Boeing 747
Rutherford	Hayes	BOE002	BOE002	Boeing 737
Mary	Jones	DC1001	DC1001	DC 10
Brad	Knickerbocker	DC1002	DC1002	DC 10
John	Smith			

Finally, after applying the predicate in the WHERE clause (descr is DC 10), two rows in the outer-join table match the search condition, producing the result.

Result table of query 2

firstname	lastname	serial_num	serial_num	descr
Mary	Jones	DC1001	DC1001	DC 10
Brad	Knickerbocker	DC1002	DC1002	DC 10

SET CONNECTION

SET CONNECTION makes the named connection the current one. Although you can establish multiple connections with the CONNECT statement, you can execute SQL statements for one (the current) connection at a time.

Format

```
EXEC SQL
    SET CONNECTION connection_name
```

Argument	Description
connection_name	(required) Name of the connection you are restoring, expressed as a character literal or a host variable. This connection must have been established by a previous CONNECT statement and must not have been terminated by a previous DISCONNECT statement.

Example

The following example makes the conn_3 connection the current one:

```
EXEC SQL
    SET CONNECTION ‘conn_3’ ;
```

UPDATE

UPDATE changes some or all values in an existing row of the specified system table or system table view. You cannot update user tables or user table views. To use UPDATE, you need DBA privilege or UPDATE privilege on all specified columns of the target system table or view.

Format

UPDATE {table_name | view_name}
SET assignment [,assignment]...
[WHERE condition]

where assignment indicates:

column_name = { expr | NULL } | (column [,column]...) = (expr [,expr]...)

Argument	Description
table_name	(required if view_name not specified) Name of the system table to be updated.
view_name	(required if table_name not specified) Name of the system table view to be updated.
SET	(required) Specifies the changes to specific column(s).
column_name/column	(required) Name of the column to be updated.
expr	(required with column_name if NULL is omitted; required with column) New value for the specified column indicated as an expression.
NULL	(required with column_name if expr is omitted) Indicates a NULL value for the column.
WHERE	(optional) Limits the range of the update according to the condition. If specified, only rows that satisfy the condition are changed. If omitted, all rows of the specified system table or view are updated.
condition	(required with WHERE) Condition limiting update range.

Example

The following UPDATE statement updates the SYSSMUSERS system table by changing the default tablespace for ACCOUNTID BOB to DEF99.

```
UPDATE SYSADM.SYSSMUSERS
SET DEFAULT_TS='DEF99'
WHERE ACCOUNTID='BOB'
```

VALUES INTO

VALUES INTO produces a result set consisting of one row and assigns the values in that row to host variables. This statement operates on values and expressions previously selected using locator variables.

Format

VALUES { expr | (expr [,expr]...) } INTO :host_variable [,:host_variable]...

Argument	Description
expr	(required) Expression that defines a single value of a one-column result set.
(expr)	(optional) One or more expressions that define the values for one or more columns of a result set.
:host_variable	(required) Host variable to which the value in the result row is assigned. If a result row contains multiple values, the first value is assigned to the first host variable in the list, the second value to the second variable, and so on.

Example

The following example uses the INSTR function to locate the start and end of a product description using a locator variable called :hv_prod_locator.

```
EXEC SQL
    VALUES (INSTR(:hv_prod_locator,'<Description>'))
    INTO :hv_start_descr ;

EXEC SQL
    VALUES (INSTR(:hv_prod_locator,'</Description>'))
    INTO :hv_end_descr ;
```

WHENEVER

WHENEVER specifies an action for three SQL runtime exceptions:

- NOT FOUND
- SQLERROR
- SQLWARNING

Depending on the exception, you can tell the program to continue with the next statement, branch to a host language label, or stop execution. You can code multiple WHENEVER statements for the same exception. WHENEVER applies until the next WHENEVER statement for the same exception or until the end of the ESQL source file. In other words, each WHENEVER overrides the previous WHENEVER statement specified for the same exception.

Note: FileTek recommends that you use CONTINUE or GOTO instead of STOP. Also, don't use NOT FOUND with array FETCH.

Format

```
EXEC SQL
    WHENEVER exception_sp action_sp
```

where exception_sp is defined as:

```
{NOT FOUND | SQLERROR | SQLWARNING}
```

and action_sp is defined as:

```
{STOP | CONTINUE | GOTO host_language_label
```

Argument	Description
exception_sp	(required) One of three exception conditions:
NOT FOUND	sqlcode is set to 100 (SQL_NOT_FOUND).
SQLERROR	sqlcode is set to negative.
SQLWARNING	sqlwarn[0] is set to W.
action_sp	(required) One of the following specific actions:
STOP	Terminate the program without any final reporting.
CONTINUE	Ignore the specified exception and continue executing the next program statement. CONTINUE is the default for each exception.
GOTO	Branch to the statement corresponding to the host_language_label.
host_language_label	The specific program label to which you're branching.
	Host language rules determine the correct use of WHENEVER with GOTO host_language_label. This label must be within the scope of all SQL statements for which the GOTO host_language_label action is active. The GOTO host_language_label action is active starting from the corresponding WHENEVER until another WHENEVER statement for the same exception or until the end of the ESQL source file.

Example

The following example uses WHENEVER to ignore the specified exception and continue with the next program statement:

```
EXEC SQL
    WHENEVER SQLERROR GOTO err ;
...
EXEC SQL
    UPDATE sysadm.syssmusers
    SET default_ts = 'juneaccount'
    WHERE accountid = 'user1' ;
...
```

FileTek Proprietary and Confidential


```
err:
EXEC SQL
    WHENEVER SQLERROR CONTINUE ;

EXEC SQL
    ROLLBACK WORK ;
```

Chapter 5

StorHouse SQL predicates

This chapter describes the following predicates supported by StorHouse:

- Basic predicate
- Complex predicate
- Quantified predicate
- BETWEEN
- EXISTS
- IN
- LIKE
- NULL

About StorHouse SQL predicates

Predicates reduce the number of rows returned by a query. You use them to set conditions within the WHERE clause of SELECT and DELETE statements. These conditions can be true or false for any row of the table. The query returns only those rows for which the predicate is true.

For example, the following SQL statement selects the ORDER_NO column from the CUSTOMER table, but only for rows that contain JOHN in the NAME column.

```
SELECT ORDER_NO
FROM CUSTOMER
WHERE NAME = 'JOHN'
```

The WHERE clause checks each row in the CUSTOMER table and returns only the rows that meet the condition NAME = 'JOHN'.

Predicate order

The optimizer typically processes predicates in the order you code them in an SQL statement. This allows some user control of which predicate is used first; so try to code the most selective predicate first. The optimizer overrides the user-specified order only when it knows that another predicate is more selective.

Comparisons of CHAR and VARCHAR fields with blanks

If a predicate compares a CHAR field (column, literal, or function value) to a VARCHAR field containing trailing blanks, and the two fields match exactly or match exactly except for additional trailing blanks in the VARCHAR field, the result is “greater than” rather than “equal.” Currently, comparison semantics trim trailing blanks from CHAR fields but not from VARCHAR fields.

For example, assume you’re comparing two fields that match exactly: a CHAR operand with one trailing blank to a VARCHAR operand with one trailing blank.

'FIELD<blank>' to 'FIELD<blank>'

StorHouse/RM trims the blank from the CHAR operand but not from the VARCHAR operand; therefore, the result is “greater than.”

'FIELD' to 'FIELD<blank>'

Additionally, StorHouse/RM treats literals as CHAR fields, so a comparison between a literal and a VARCHAR field containing trailing blanks always produces an “unequal” result. Use of a VARCHAR host variable, however, does produce an “equal” result when compared to a literal, provided that the number of trailing blanks match.

Basic predicate

A *basic predicate* compares two values with a relational operator. If any expression in the predicate evaluates to NULL, the result is UNKNOWN.

Format

[NOT] basic_pred [AND | OR basic_pred]

where basic_pred is defined as:

expr1 rel_op {expr2 | query_expression}

Argument	Description
expr1	(required) First value in the comparison expressed as an identifier or a value.
rel_op	(required) Relational operator (also known as a comparison operator) that compares two values. Valid operators are: = two equal values > first value greater than second < first value less than second >= first value greater than or equal to second <= first value less than or equal to second <> first value not equal to second
expr2	(required if you omit query_expression) Second value in the comparison expressed as an identifier or a value.
query_expression	(required if you omit expr2) Second value in the comparison expressed as a query. A query expression must return only one value.

Example

The following SELECT statement, using the basic predicate COUNTRY <> ‘USA’, selects the NAME column from the CUSTOMER table. This predicate returns only those rows where COUNTRY is not equal to USA.

```
SELECT NAME
FROM CUSTOMER
WHERE COUNTRY <> ‘USA’
```

Complex predicate

A *complex predicate* contains logical (or boolean) operators that relate one or more predicates and produce a true, false, or unknown value. StorHouse recognizes three logical operators: NOT, AND, and OR.

Operator	Description
NOT	Takes a predicate as its argument and changes its value from false to true or true to false. Example: NOT(NAME = ‘JOHN’) returns only rows that do not contain JOHN in the NAME column.
AND	Takes two predicates as arguments and evaluates them as true only if both predicates are true. Example: NAME = ‘JOHN’ AND CITY = ‘CHICAGO’ returns only rows that contain JOHN in the NAME column and CHICAGO in the CITY column.
OR	Takes two predicates as arguments and evaluates them as true if either one of the predicates is true. Example: NAME = ‘JOHN’ OR CITY = ‘CHICAGO’ returns rows that contain JOHN in the NAME column or rows that contain CHICAGO in the CITY column.

When you include multiple logical operators, StorHouse/RM evaluates conditions within parentheses first. If you don’t include parentheses, then StorHouse/RM evaluates NOT conditions before AND and AND conditions before OR. StorHouse/RM optimizes conditions with the same logical operator, for instance, AND and AND.

See “Logical operators (AND, OR, NOT)” on page 2-20 for more information about the results of OR and AND operators.

Examples

- In the first complex predicate that follows, StorHouse/RM applies AND before OR. In the second, StorHouse/RM applies OR before AND.

  SALARY > SSN AND COMM > CC OR BONUS > BB

  SALARY > SSN AND (COMM > CC OR BONUS > BB)

- In the first complex predicate that follows, StorHouse/RM applies NOT before AND. In the second, StorHouse/RM applies AND before NOT.

  NOT SALARY > SSN AND COMM > CC

  NOT (SALARY > SSN AND COMM > CC)

- In the following complex predicate, StorHouse/RM applies AND first. StorHouse/RM can select the order of applying the ORs without changing the result.

  SALARY > SSN AND COMM > CC OR BONUS > BB OR SEX = FF

Quantified predicate

A *quantified predicate* compares a value to a collection of values preceded by the keyword ANY or SOME. The predicate evaluates to true if the specified relationship is true for at least one value returned by the query expression. The predicate evaluates to false if the query expression returns no values or the specified relationship is false for all returned values.

Format

[NOT] quant_pred [AND | OR quant_pred]

where quant_pred is defined as:

expr rel_op {ANY | SOME} (query_expression)

Argument	Description
expr	(required) First value in the comparison expressed as an identifier or a value.
rel_op	(required) Relational operator (also known as a comparison operator) that compares two values. Valid operators are: = two equal values > first value greater than second < first value less than second >= first value greater than or equal to second <= first value less than or equal to second <> first value not equal to second
query_expression	(required) Values in the comparison expressed as a query.

Example

The following SELECT statement, using the quantified predicate ANY, selects the NAME column from the CUSTOMER table where COUNTRY equals any of the countries in the SUPPLIER table.

```
SELECT NAME
FROM CUSTOMER
WHERE COUNTRY = ANY
(SELECT COUNTRY
FROM SUPPLIER)
```

BETWEEN

BETWEEN determines whether a value falls within a specified range. The first expression specifies the lower limit, and the second expression specifies the upper limit. The predicate evaluates to true if the value falls within the specified range or is equal to either the lower or upper limit.

Format

[NOT] between_pred [AND | OR between_pred]

where between_pred is defined as:

expr [NOT] BETWEEN expr1 AND expr2

Argument	Description
expr	(required) First value in the comparison expressed as an identifier or a value.
expr1	(required) Lower limit of the range expressed as an identifier or a value.
expr2	(required) Upper limit of the range expressed as an identifier or a value.

Example

The following SELECT statement, using the BETWEEN predicate, selects the JOB_TITLE column from the EMPLOYEE table where the value in the SALARY column falls between 2000.00 and 10000.00.

```
SELECT JOB_TITLE
FROM EMPLOYEE
WHERE SALARY BETWEEN 2000.00 AND 10000.00
```

EXISTS

EXISTS checks for the existence of specific rows. In this case, the query expression returns entire rows rather than values. The predicate evaluates to true if the number of rows returned by the query expression is not equal to zero.

Format

[NOT] exists_pred [AND | OR exists_pred]

where exists_pred is defined as:

EXISTS (query_expression)

Argument	Definition
query_expression	(required) Query that defines the rows you want to check.

Example

The following statement, using the EXISTS predicate, evaluates to true if the CUST_ID column in the ORDER_TBL table has any rows that contain the CUSTID 4531.

```
EXISTS (SELECT * FROM ORDER_TBL
WHERE CUSTID = 4531)
```

IN

IN checks whether a specified value falls within a stated set of values.

Format

[NOT] in_pred [AND | OR in_pred]

where in_pred is defined as:

expr [NOT] IN (query_expression) | (:host_variable | literal)
[,(:host_variable | literal)]...

Argument	Description
expr	(required) Value you want to check defined as an identifier or a value.
query_expression	(required if you omit a literal) Query that defines the set of values.
:host_variable	(required if you omit a query_expression and literal) Host language variable referred to on an SQL statement.
literal	(required if you omit a query_expression and :host_variable) Constant that defines a member of the value set.

Example

The following statement, using the IN predicate, checks whether any values in the STATE column of the ADDRESS table equal MA or NH. The predicate evaluates to true if at least one row contains MA or NH.

ADDRESS.STATE IN (‘MA’, ‘NH’)

LIKE

LIKE searches a specified column for strings that contain a specified pattern. An underscore (_) in the pattern matches any single character of the string. A percent sign (%) in the pattern matches zero or more characters of the string. You can use the optional ESCAPE clause to disable the special meaning given to the underscore and percent sign.

The LIKE predicate, when applied to an indexed column, can use an index scan instead of a full table scan. StorHouse can scan range indexes and value indexes. Those indexes can be simple indexes or compound indexes with the column as the first field in the index.

LIKE does not ignore trailing blanks in the column value. To be matched, the pattern string must contain matching trailing blanks, or a matching number of trailing '_' characters, or a trailing '%" character.

Note: If you specify NOT LIKE or LIKE with leading wildcard characters (for example, LIKE '%ABC'), StorHouse must sequentially read the entire index.

Format

[NOT] like_pred [AND | OR like_pred]

where like_pred is defined as:

column_name [NOT] LIKE {string_constant | :host_variable}
[ESCAPE character_constant1]

Argument	Description
column_name	(required) Name of the column to be searched for the specified string. Type must be a CHAR type.
string_constant	(required if you don't specify :host_variable) String constant that contains the search pattern.

Argument	Description
:host_variable	(required if you don't specify string_constant) Host variable that contains the search pattern. The host variable must be a character string type variable.
character_constant1	(required with optional ESCAPE clause) Character to be used as the ESCAPE character.

Examples

- The following statement, using the LIKE predicate, evaluates to true for all values in the CUST_NAME column that contain the literal string COMPUTER.

  CUST_NAME LIKE '%COMPUTER%'

- The following statement, using the LIKE predicate, evaluates to true for all values in the CUST_NAME column that contain exactly three characters.

  CUST_NAME LIKE '_ _ _'

- The following statement, using the LIKE predicate, uses the backslash (\) to specify the ESCAPE character that disables the special interpretation given to the underscore. This predicate evaluates to true for all rows in the ITEM_NAME column that contain underscore characters.

  ITEM_NAME LIKE '%_%' ESCAPE '\'

NULL

NULL checks the specified column for NULL values. You can specify NOT to test for non-NULL values.

Format

[NOT] null_pred [AND | OR null_pred]

where null_pred is defined as:

column_name IS [NOT] NULL

Argument	Description
column_name	(required) Name of the column to be tested.

Examples

- The following SELECT statement, using the NULL predicate, selects the ORDER_NO column from the ORDERS table where the CONTACT_NAME column contains a NULL value.

  ```
  SELECT ORDER_NO
  FROM ORDERS
  WHERE CONTACT_NAME IS NULL
  ```

- The following SELECT statement, using the NULL predicate with the NOT keyword, selects the NAME column from the CUSTOMER table where the LAST_ORDER column does not contain a NULL value.

  ```
  SELECT NAME
  FROM CUSTOMER
  WHERE LAST_ORDER IS NOT NULL
  ```

Chapter

6

StorHouse SQL functions

This chapter contains formats and examples of the aggregate and scalar functions supported by StorHouse. Functions are described in alphabetical order.

About StorHouse SQL functions

A *function* is a named operation followed by one or more arguments in parentheses. You use functions to map a group of values to a single value. StorHouse supports two types of functions: aggregate and scalar. You can use both types of functions in SELECT statements.

Aggregate functions

An *aggregate function*, also called *column function*, is an SQL operation that derives its result from a collection of values across one or more rows. If the statement contains a GROUP BY clause, the aggregate function returns one value for each group. Otherwise, StorHouse/RM treats the result of the entire SELECT statement as one group. You cannot nest aggregate functions (for example, average a count).

The following table lists the StorHouse aggregate functions.

Aggregate function	Description	Page
AVG	Computes the average of a set of numbers	6-9
COUNT	Counts the number of rows or non-NULL values	6-18
MAX	Returns the highest value in a column or expression	6-37
MIN	Returns the lowest value in a column or expression	6-38
SUM	Calculates the sum of all values in a column or expression	6-55

Scalar functions

A *scalar function* is an SQL operation that produces a single value from another value. You express a scalar function as a function name followed by a list of arguments enclosed in parentheses. The following table lists the StorHouse scalar functions:

Scalar function	Description	Page
ABS	Computes an absolute value of an expression	6-5
ADD_MONTHS	Adds a specified number of months to a date	6-7
ASCII	Returns the ASCII value of the first character	6-8
BIT_LENGTH	Returns the length (in bits) of an expression	6-11
BLOB	Returns a BLOB representation of an expression	6-12
CHAR_LENGTH	Returns the length (in characters) of an expression	6-13
CHR	Returns a character string for an integer expression	6-14
CLOB	Returns a CLOB representation of a character string	6-15
CONCAT	Concatenates two expressions	6-16

Scalar function	Description	Page
DAYOFMONTH	Returns the day of the month for a date expression	6-19
DAYOFWEEK	Returns the day of the week for a date expression	6-20
DAYOFYEAR	Returns the day of the year for a date expression	6-21
DAYS	Returns an integer value equal to the number of days since 1/1/0001 plus 1.	6-22
DECODE	Compares a value to criteria and returns values according to a match expression	6-23
GREATEST	Returns the greatest value of an expression	6-25
HOUR	Returns the hour (0–24) of a time expression	6-26
INITCAP	Converts the first character of character expression to uppercase and subsequent characters to lowercase	6-27
INSTR	Searches for an expression and returns its position	6-28
LAST_DAY	Returns the date of the last day of the month in a date expression	6-30
LEAST	Returns the lowest value of specified expressions	6-31
LENGTH	Calculates the length of a value in an expression	6-32
LOWER	Converts a character string to lowercase	6-33
LPAD	Pads to the left (beginning) of a character string	6-34
LTRIM	Trims leading characters from a character string	6-36
MINUTE	Returns the minute value in a time expression	6-39
MONTH	Returns the month value in a date expression	6-40
MONTHS_BETWEEN	Computes the number of months between two dates	6-41
NEXT_DAY	Calculates the date of the next occurrence of the specified day of the week after the date expression	6-42
NVL	Returns the value of the first expression; but if NULL, returns the value of the second expression	6-43
OCTET_LENGTH	Returns the length (in bytes) of an argument	6-44

Scalar function	Description	Page
OVERLAY	Replaces a substring from the first argument with the second argument	6-45
POSITION	Determines the starting position at which the first expression is found in the second expression	6-47
QUARTER	Determines the quarter of the year in which a date occurred	6-48
RPAD	Pads to the right (ending) of a character string	6-49
RTRIM	Removes ending characters from a character string	6-51
SECOND	Returns the number of seconds in a time expression	6-52
SUBSTR	Returns a part of an expression based on a starting position and a length	6-53
TO_CHAR	Converts an expression to a character value	6-57
TO_DATE	Converts a character expression to a date value	6-58
TO_HEX	Converts a BINARY, BLOB, or VARBINARY column value to a character value	6-59
TO_NUMBER	Converts a character expression to a number value	6-61
TO_TIME	Converts a character expression to a time value	6-62
TRANSLATE	Translates each character in a character expression	6-63
TRIM	Removes leading values, trailing values, or both	6-65
UPPER	Converts a character expression to uppercase	6-67
WEEK	Returns the week of a year in a date expression	6-68
YEAR	Returns the year in a date expression	6-69

ABS

The scalar function ABS computes the absolute value of a specified expression. If the expression evaluates to NULL, the result is NULL. The result data type is as follows:

If the input type is	The the result type is
REAL	DOUBLE PRECISION
DOUBLE PRECISION	DOUBLE PRECISION
SMALLINT	INTEGER
INTEGER	INTEGER
NUMERIC	NUMERIC (31,8)

Format

ABS (expression)

Argument	Description
expression	(required) Identifier or value for which you want the absolute value calculated. The data type must be SMALLINT, INTEGER, NUMERIC, REAL, or DOUBLE PRECISION.

Example

The following SELECT statement, using the scalar function ABS, selects the absolute value of the months between the current (system) date and the date in the ORDER_DATE column of the ORDERS table, but only where the absolute value is greater than 3.

```
SELECT ABS (MONTHS_BETWEEN (SYSDATE, ORDER_DATE))
FROM ORDERS
WHERE ABS (MONTHS_BETWEEN (SYSDATE, ORDER_DATE)) > 3
```

ADD_MONTHS

The scalar function ADD_MONTHS adds a specified number of months to a specified date. The result data type is DATE. If any argument evaluates to NULL, the result is NULL. Note that when the integer_expression is negative, you will actually subtract from the date.

Format

ADD_MONTHS (date_expression, integer_expression)

Argument	Description
date_expression	(required) Date to which you want to add months. The data type must be DATE.
integer_expression	(required) Number of months you want to add to the date_expression. The data type must be INTEGER.

Example

The following SELECT statement, using the scalar function ADD_MONTHS, selects all rows from the CUSTOMER table, where the date in the START_DATE column plus six months is greater than the current (system) date.

```
SELECT *
FROM CUSTOMER
WHERE ADD_MONTHS (START_DATE, 6) > SYSDATE
```

ASCII

The scalar function ASCII returns the ASCII value of the first character of the specified character expression. The result data type is SMALLINT. If the character expression evaluates to NULL, the result is NULL.

Format

ASCII (char_expression)

Argument	Description
char_expression	(required) Expression for which you want to calculate the ASCII value of the first character. The data type must be CHAR, CLOB, or VARCHAR.

Example

The following SELECT statement, using the scalar function ASCII, calculates the ASCII value of the first character of the ZIP column from the CUSTOMER table.

```
SELECT ASCII (ZIP)
FROM CUSTOMER
```

AVG

The aggregate function AVG computes the average of a set of numbers. StorHouse/RM eliminates NULL values before computing the average. If all values are NULL, the result is NULL. The result data type is as follows:

If the input type is	Then the result type is
REAL	DOUBLE PRECISION
DOUBLE PRECISION	DOUBLE PRECISION
SMALLINT	DECIMAL(13,8)
INTEGER	DECIMAL(18,8)
DECIMAL(p1,s1)	s = MIN (31 - p1, 8) + s1 p = p1-s1 + s

Format

AVG ({[ALL] expression} | {DISTINCT column_ref})

Argument	Description
ALL	(optional) Includes duplicate values in the calculation.
DISTINCT	(optional) Excludes duplicate values in the calculation.
expression	(required if you omit column_ref) Arithmetic expression that contains at least one column_name and does not contain another aggregate function specification. The data type must be SMALLINT, INTEGER, DECIMAL, REAL, or DOUBLE PRECISION.
column_ref	(required if you omit expression) Name of the column whose average will be calculated. The column data type must be SMALLINT, INTEGER, DECIMAL, NUMERIC, REAL or DOUBLE PRECISION.

Example

The following SELECT statement, using the aggregate function AVG, calculates the average salary of all employees in department number 20.

```
SELECT AVG (SALARY)
FROM EMPLOYEE
WHERE DEPTNO = 20
```

BIT_LENGTH

The scalar function BIT_LENGTH returns the length, in bits, of an expression. The result data type is DECIMAL (19,8). If the expression is NULL, the result is NULL.

Format

BIT_LENGTH (expression)

Argument	Description
expression	(required) Expression to be searched. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.

Example

The following SELECT statement, using the scalar function BIT_LENGTH, returns the length of the column NAME in bits.

```
SELECT BIT_LENGTH(NAME)
FROM CUSTOMER
```

BLOB

The scalar function BLOB returns a BLOB representation of a string. The result data type is BLOB. If the expression is NULL, the result is NULL.

Format

BLOB (expression [, length])

Argument	Description
expression	(required) Expression to be returned as a BLOB representation of a string. The data type must be BINARY, CHAR, VARBINARY, or VARCHAR.
length	(optional) Length of the resulting string. Include a length to limit the result to the specified length.

Example

The following SELECT statement, using the scalar function BLOB, allows for comparison against a BLOB column.

```
SELECT *
FROM BLOB_DATA
WHERE BLOB_COLUMN = BLOB(X'12345678')
```

CHAR_LENGTH

The scalar function CHAR_LENGTH returns the length, in characters, of an expression. The result data type is INTEGER. If the expression is NULL, the result is NULL.

Format

CHAR_LENGTH (expression)

Argument	Description
expression	(required) Expression to be searched. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.

Example

The following SELECT statement, using the scalar function CHAR_LENGTH, selects all names greater than 10 characters from the CUSTOMER table.

```
SELECT NAME
FROM CUSTOMER
WHERE CHAR_LENGTH(NAME) > 10
```

CHR

The scalar function CHR takes an integer expression as an argument and returns a character string whose first character has the ASCII value equal to the integer expression. The result data type is CHAR. If the integer expression evaluates to NULL, the result is NULL.

Format

CHR (integer_expression)

Argument	Description
integer_expression	(required) Number you want to convert to a character. The data type must be INTEGER or SMALLINT.

Example

The following SELECT statement, using the scalar function CHR, selects all the values in the CUSTOMER table where the first character of the zip code is equal to the ASCII value 53.

```
SELECT *
FROM CUSTOMER
WHERE SUBSTR (ZIP, 1, 1) = CHR (53)
```

CLOB

The scalar function CLOB returns a CLOB representation of a character string. The result data type is CLOB. If the expression is NULL, the result is NULL.

Format

CLOB (char_expression [, length])

Argument	Description
char_expression	(required) Expression to be returned as a CLOB representation of a string. The data type must be CHAR or VARCHAR.
length	(optional) Length of the resulting string. Include a length to limit the result to the specified length.

Example

The following SELECT statement, using the scalar function CLOB, forces a column to be returned in the data type CLOB.

```
SELECT CLOB(NAME) FROM CUSTOMER
```

CONCAT

The scalar function CONCAT returns the concatenation of two compatible string arguments. You can also use the concatenation operator (||) in place of the CONCAT keyword. Note the following:

- If either of the expressions is CLOB, the result is CLOB.
- If either of the expressions is BLOB, the result is BLOB.
- Otherwise, if the first expression is CHAR or VARCHAR, the result is a character string of VARCHAR data type.
- Or if the first expression is BINARY or VARBINARY, the result is a binary string of VARBINARY data type.

Trailing blanks are preserved in CHAR input expressions. If one of the expressions evaluates to NULL, then the result is NULL. If one of the expressions is a zero-length string, then the result is the other expression. The resulting length is the sum of the lengths of the two arguments. If this length is greater than the maximum length of the result data type, StorHouse/RM trims the result.

Format

CONCAT (expression1, expression2)

Argument	Description
expression1	(required) First argument (expressed as an identifier or value) to concatenate to the second argument. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.
expression2	(required) Second argument (expressed as an identifier or value) to concatenate to the first argument. The data type must be compatible with the first argument, that is, both character types or both binary types.

Examples

- The following SELECT statement, using the scalar function CONCAT, selects names, employee numbers, and salaries from the CUSTOMER table where the PROJECT is equal to the PROJ_NAME preceded by the characters US.

  ```
  SELECT NAME, EMPNO, SALARY
  FROM CUSTOMER
  WHERE PROJECT = CONCAT('US', PROJ_NAM)
  ```

- The following concatenation operation returns Hello World!

  ‘Hello ’ || ‘World!’

COUNT

The aggregate function COUNT computes either the number of rows in a table or the number of non-NULL values in a column or group of values. The result data type is INTEGER, and it cannot have a NULL value.

Format

COUNT ({ [ALL] expression} | [DISTINCT] column_ref | * })

Argument	Description
ALL	(optional) Includes duplicate values in the count.
expression	(required if you omit column_ref or *) Argument to be counted expressed as either an identifier or a value. Any data type is valid.
DISTINCT	(optional) Eliminates duplicate values from the count. You cannot use the DISTINCT option if the column_ref is a BLOB or CLOB data type.
column_ref	(required if you omit expression or *) Name of the column to be counted. Any data type is valid.
*	(required if you omit expression or column_ref) Indicates that the function will count the number of rows in a table.

Example

The following SELECT statement, using the aggregate function COUNT, counts the number of rows that have an order date equal to the current (system) date.

```
SELECT COUNT (*)
FROM ORDERS
WHERE ORDER_DATE = SYSDATE
```

DAYOFMONTH

The scalar function DAYOFMONTH returns the day of the month in the argument as a SMALLINT ranging from 1 to 31. The result data type is SMALLINT. If the date expression evaluates to NULL, the result is NULL.

Format

DAYOFMONTH (date_expression)

Argument	Description
date_expression	(required) Date you want expressed as a day-of-the-month (1-31) value. This data type must be DATE.

Example

The following SELECT statement, using the scalar function DAYOFMONTH, selects rows from the ORDERS table where the order date is the fourteenth day of the month.

```
SELECT *
FROM ORDERS
WHERE DAYOFMONTH (ORDER_DATE) = 14
```

DAYOFWEEK

The scalar function DAYOFWEEK returns the day of the week in the argument as a SMALLINT ranging from 1 to 7. The result data type is SMALLINT. If the date expression evaluates to NULL, the result is NULL.

Format

DAYOFWEEK (date_expression)

Argument	Description
date_expression	(required) Date you want expressed as a day-of-the-week value (1-7; where Sunday is 1, Monday is 2, and so on). The data type must be DATE.

Example

The following SELECT statement, using the scalar function DAYOFWEEK, selects all rows in the ORDERS table with an order date of Monday.

```
SELECT *
FROM ORDERS
WHERE DAYOFWEEK (ORDER_DATE) = 2
```

DAYOFYEAR

The scalar function DAYOFYEAR returns the day of the year in the argument as a SMALLINT ranging from 1 to 366. The result data type is SMALLINT. If the date expression evaluates to NULL, the result is NULL.

Format

DAYOFYEAR (date_expression)

Argument	Description
date_expression	(required) Date you want expressed as a day-of-the-year value. The data type must be DATE.

Example

The following SELECT statement, using the scalar function DAYOFYEAR, selects all rows in the ORDERS table where the order date is the 300th day of the year.

```
SELECT *
FROM ORDERS
WHERE DAYOFYEAR (ORDER_DATE) = 300
```

DAYS

The scalar function DAYS returns an integer representation of a date. The single input argument must be a DATE, a TIMESTAMP, or a CHAR or VARCHAR that contains a valid string representation of a date or timestamp. The result is an integer value equal to the number of days since 1/1/0001, plus 1. If the argument is NULL, the result is a NULL value.

Format

DAYS (date_expression)

Argument	Description
date_expression	(required) Date you want expressed as the number of days since 1/1/0001 plus 1. The data type must be DATE, TIMESTAMP, CHAR, or VARCHAR.

Example

The following SELECT statement, using the scalar function DAYS, returns the number of days from the date that employee JSMITH was hired to the current date.

```
SELECT DAYS (SYSDATE) – DAYS (HIREDATE)
FROM EMPLOYEES
WHERE EMP_NAME = 'JSMITH'
```

DECODE

The scalar function DECODE compares the value of the first expression with one or more search expressions. For matches, DECODE returns the corresponding match expression. For no matches, DECODE returns the specified default expression. The result type is the same as that of the first match expression. For no match conditions where the default expression is omitted, DECODE returns a NULL value.

Format

DECODE (expression, search_expression, match_expression
[, search_expression, match_expression]...
[, default_expression])

Argument	Description
expression	(required) First argument in the comparison expressed as an identifier or a value. Any data type except BLOB or CLOB is valid.
search_expression	(at least one is required) Arguments in the comparison expressed as an identifier or a value. The data type of each search_expression must be compatible with the expression data type.
match_expression	(at least one is required) Identifier or value that is returned when expression and search_expression compare equally. Any expression type is valid; however, all match_expressions must have the same data type within a DECODE function.
default_expression	(optional) Default expression expressed as an identifier or a value. The default_expression data type must be compatible with expression data type.

Example

The following SELECT statement, using the scalar function DECODE, selects all employees from the EMPLOYEE table and assigns them department names according to their department number (DEPTNO) values. For example, all employees with a department number of 20 are assigned to the RESEARCH department.

```
SELECT ENAME,
DECODE (DEPTNO,
10, 'ACCOUNTS',
20, 'RESEARCH',
30, 'SALES',
40, 'SUPPORT',
'NOT ASSIGNED')
FROM EMPLOYEE
```

When an employee's department number is not equal to 10, 20, 30, or 40, that employee is considered "NOT ASSIGNED."

GREATEST

The scalar function GREATEST returns the greatest value of the specified expressions. The result data type is the same type as the first expression. If any expression evaluates to NULL, the result is NULL.

Format

GREATEST (expression [,expression]...)

Argument	Description
expression	(at least one is required) An identifier or a value to be included in the comparison of values. The first expression can be any data type except BLOB or CLOB. All subsequent expressions must be the same data type as the first.

Example

The following SELECT statement, using the scalar function GREATEST, returns all customer numbers and names from the CUSTOMER table with the later of the last date a payment was made and the last time the customer called customer services.

```
SELECT CUST_NO, NAME,
GREATEST (LAST_PYMT_DATE, LAST_CALL_DATE)
FROM CUSTOMER
```

HOUR

The scalar function HOUR returns the hour in the argument as a SMALLINT ranging from 0 to 24. The result data type is SMALLINT. If the time expression evaluates to NULL, the result is NULL.

Format

HOUR (time_expression)

Argument	Description
time_expression	(required) Time you want expressed as an hour-of-the-day (0-24) value. The data type must be TIME or TIMESTAMP.

Example

The following SELECT statement, using the scalar function HOUR, selects all rows from the ARRIVALS table when the time of arrival (IN_TIME) is before 12:00 P.M.

```
SELECT *
FROM ARRIVALS
WHERE HOUR (IN_TIME) < 12
```

INITCAP

The scalar function INITCAP converts the first character of a character expression to uppercase and subsequent characters in that character expression to lowercase. The result data type is the same as the argument, that is CHAR, VARCHAR, or CLOB. If the character expression evaluates to NULL, the result is NULL.

Format

INITCAP (char_expression)

Argument	Description
char_expression	(required) Identifier or value to be converted to initial capitals. The data type must be CHAR, CLOB, or VARCHAR.

Example

The following SELECT statement, using the scalar function INITCAP, converts all names in the CUSTOMER table to initial capitals and converts subsequent characters to lowercase.

```
SELECT INITCAP (NAME)
FROM CUSTOMER
```

INSTR

The scalar function INSTR searches within the first specified expression for the second specified expression and then returns the starting position of the first occurrence of expression1 within expression2. The result data type is INTEGER. If the search is unsuccessful, the INSTR function returns zero. If either argument evaluates to NULL, the result is NULL.

Note: The POSITION function performs a similar operation.

Format

INSTR (expression1, expression2 [, start_position [, occurrence]])

Argument	Description
expression1	(required) String to be searched, expressed as an identifier or a value. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.
expression2	(required) String to be found, expressed as an identifier or a value. The data type must be compatible with expression1, that is, if expression1 is a CHAR (character string) then expression2 must be CHAR or VARCHAR. If expression1 is CLOB, then expression2 must be CHAR, VARCHAR, or CLOB. If expression1 is BLOB, then expression2 must be BINARY, VARBINARY, or BLOB.
start_position	(optional) Location in expression1 where the search is to start. A 1 indicates the first character or byte, 2 the second character or byte, and so on. If specified, start_position must be an INTEGER type. If omitted, the search begins at the first character or byte of expression1.
occurrence	(optional) Number of occurrences to be found, where 1 is the first, 2 is the second, and so on. If specified, the type must be INTEGER. If omitted, INSTR searches for the first occurrence only.

Example

The following SELECT statement, using the scalar function INSTR, selects the customer number and name from all the rows in the CUSTOMER table where the string 'heritage' appears somewhere in the address column.

```
SELECT CUST_NO, NAME
FROM CUSTOMER
WHERE INSTR (ADDR, 'heritage') > 0
```

The value returned by INSTR contains the actual locations of the string in the address column. The position must be greater than zero because zero represents a not found condition. The comparison is always case sensitive unless you use LOWER or UPPER on expression1.

LAST_DAY

The scalar function LAST_DAY returns the date of the last day of the month containing the argument date. The result data type is DATE. If the argument evaluates to NULL, the result is NULL.

Format

LAST_DAY (date_expression)

Argument	Description
date_expression	(required) Date for which you want to find the last day of the month. The data type must be DATE.

Example

The following SELECT statement, using the scalar function LAST_DAY, returns all orders from the ORDERS table for the month of August.

```
SELECT *
FROM ORDERS
WHERE LAST_DAY (ORDER_DATE) = '08/31/1995'
```

LEAST

The scalar function LEAST returns the lowest value of the specified expressions. If any of the expressions evaluate to NULL, then the result is NULL. The result type is the same type as the first expression.

Format

LEAST (expression [,expression]...)

Argument	Description
expression	(at least one is required) One of the values to be evaluated expressed as an identifier or a value. The first expression can be any data type other than BLOB or CLOB. Subsequent expressions must be the same data type as the first.

Example

The following SELECT statement, using the scalar function LEAST, returns all customer numbers and names from the CUSTOMER table with the earlier of the date the last payment was made and the last time the customer called customer services.

```
SELECT CUST_NO, NAME,
LEAST (LAST_PYMT_DATE, LAST_CALL_DATE)
FROM CUSTOMER
```

LENGTH

The scalar function LENGTH returns the length of the value of the specified expression. The result data type is INTEGER. If the expression evaluates to NULL, the result is NULL.

Format

LENGTH (expression)

Argument	Description
expression	(required) Expression whose length is to be calculated. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.

Example

The following SELECT statement, using the scalar function LENGTH, selects all names greater than 5 characters from the CUSTOMER table.

```
SELECT NAME 'LONG NAME'
FROM CUSTOMER
WHERE LENGTH (NAME) > 5
```

LOWER

The scalar function LOWER takes a specified character expression, converts it to lowercase, and returns the resulting character string. The result data type is the same as the expression, that is CHAR, CLOB, or VARCHAR. If the character expression evaluates to NULL, the result is NULL.

Format

LOWER (char_expression)

Argument	Description
char_expression	(required) Character expression to be converted to lowercase. The data type must be CHAR, CLOB, or VARCHAR.

Example

The following SELECT statement, using the scalar function LOWER, converts the NAME column values in the CUSTOMER table to lowercase for comparison with the value smith, and then returns all matching names.

```
SELECT NAME
FROM CUSTOMER
WHERE LOWER (NAME) = 'smith'
```

LPAD

The scalar function LPAD pads the beginning of the character string specified in the first argument with the character string specified in the third argument. The length of the resulting character string is specified in the second argument. The result data type is the same as the expression, that is, CHAR, CLOB, or VARCHAR. If the character string evaluates to NULL, the result is NULL.

LPAD works as follows:

If first argument is	Then
< specified length	It's padded until it reaches the specified length.
= specified length	The resulting string is the same length as the first argument.
> specified length	It's truncated to the specified length.

Format

LPAD (char_expression, length [, pad_expression])

Argument	Description
char_expression	(required) Character string to be padded expressed as an identifier or a value. The data type must be CHAR, CLOB, or VARCHAR.
length	(required) Length of the resulting character string (after padding). The data type must be INTEGER.
pad_expression	(optional) Character used to pad char_expression expressed as an identifier or a value. If specified, the data type must be CHAR. If omitted, char_expression is padded with blanks.

Examples

- The following SELECT statement, using the scalar function LPAD, pads the beginning of each character string found in the NAME column of the CUSTOMER table with blanks until it reaches 30 characters in length.

  ```
  SELECT LPAD (NAME, 30)
  FROM CUSTOMER
  ```

- The following SELECT statement, using the scalar function LPAD, pads the beginning of each character string found in the NAME column of the CUSTOMER table with periods until it reaches 30 characters in length.

  ```
  SELECT LPAD (NAME, 30, '.')
  FROM CUSTOMER
  ```

LTRIM

The scalar function LTRIM removes all specified leading characters or bytes from a given expression and returns the resulting expression. The result data type is the same as the expression type. If the expression evaluates to NULL, the result is NULL.

Format

LTRIM (expression, char_set)

Argument	Description
expression	(required) Expression whose leading characters or bytes will be deleted, expressed as an identifier or a value. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.
char_set	(required) Leading characters or bytes to be deleted from the expression. If the expression is a character type (CHAR, VARCHAR, CLOB) then the char_set must be a character type (including a character literal). If the expression is a binary type (BINARY, VARBINARY, BLOB) then the char_set must be a binary type (including a hex literal). The default char_set is ' ' (blank) for character types and x'00' for binary types.

Example

The following SELECT statement, using the scalar function LTRIM, selects names and addresses from the CUSTOMER table and deletes all leading spaces from the addresses.

```
SELECT NAME, LTRIM (ADDR, ' ')
FROM CUSTOMER
```

MAX

The aggregate function MAX returns the maximum value in a set of values. The result data type is the same as the expression.

Format

MAX (expression)

Argument	Description
expression	(required) Group of values included in the comparison expressed as an identifier or a value. The expression can be any data type except BLOB or CLOB.

Example

The following SELECT statement, using the aggregate function MAX, selects the largest order amount for each product by order date from the ORDERS table.

```
SELECT ORDER_DATE, PRODUCT, MAX (QTY)
FROM ORDERS
GROUP BY ORDER_DATE, PRODUCT
```

MIN

The aggregate function MIN returns the minimum value in a set of values. The result data type is the same as the expression.

Format

MIN (expression)

Argument	Description
expression	(required) Group of values included in the comparison expressed as an identifier or a value. The expression can be any data type except BLOB or CLOB.

Example

The following SELECT statement, using the aggregate function MIN, selects the minimum salary in the EMPLOYEE table for employees in department number 20.

```
SELECT MIN (SALARY)
FROM EMPLOYEE
WHERE DEPTNO = 20
```

MINUTE

The scalar function MINUTE returns the minute value in the specified time expression as a SMALLINT ranging from 0 to 59. If the time expression evaluates to NULL, the result is NULL.

Format

MINUTE (time_expression)

Argument	Description
time_expression	(required) Time expression for which you want the minutes value converted to a SMALLINT in the range of 0-59. The data type must be TIME or TIMESTAMP.

Example

The following SELECT statement, using the scalar function MINUTE, selects all rows from the ARRIVALS table where the minutes portion of the IN_TIME is greater than 10 minutes.

```
SELECT *
FROM ARRIVALS
WHERE MINUTE (IN_TIME) > 10
```

MONTH

The scalar function MONTH returns the month value in the specified date as a SMALLINT ranging from 1 to 12. If the date expression evaluates to NULL, the result is NULL.

Format

MONTH (date_expression)

Argument	Description
date_expression	(required) Date for which you want the month converted to a SMALLINT in the range of 1-12. The data type must be DATE.

Example

The following SELECT statement, using the scalar function MONTH, selects all rows from the ORDERS table where the ORDER_DATE occurs in June.

```
SELECT *
FROM ORDERS
WHERE MONTH (ORDER_DATE) = 6
```

MONTHS_BETWEEN

The scalar function MONTHS_BETWEEN computes the number of months between two specified dates. The result data type is INTEGER. If either date expression evaluates to NULL, the result is NULL. The result is negative if the second date occurs before the first date.

Format

MONTHS_BETWEEN (date_expression1, date_expression2)

Argument	Description
date_expression1	(required) First date in the specified range. The data type must be DATE.
date_expression2	(required) Second date in the specified range. The data type must be DATE.

Example

The following SELECT statement, using the scalar function MONTHS_BETWEEN, calculates the number of months between the current (system) date and the order date for order number 1002 in the ORDERS table.

```
SELECT MONTHS_BETWEEN (SYSDATE, ORDER_DATE)
FROM ORDERS
WHERE ORDER_NO = 1002
```

NEXT_DAY

The scalar function NEXT_DAY calculates the calendar date of the next occurrence of the specified day of the week after the specified date expression. The result data type is DATE. If any of the arguments evaluates to NULL, the result is NULL.

Format

NEXT_DAY (date_expression, day_of_week)

Argument	Description
date_expression	(required) Date that is used as the base date for the day-of-the-week calculation expressed as an identifier or a value. The data type must be DATE.
day_of_week	(required) Day of the week whose calendar date you want to determine. The data type must be CHAR, and the value must be either SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, or SATURDAY.

Example

The following SELECT statement, using the scalar function NEXT_DAY, calculates and returns the calendar date of the first Monday occurring after each ORDER_DATE in the ORDERS table.

```
SELECT NEXT_DAY (ORDER_DATE, 'MONDAY')
FROM ORDERS
```

NVL

The scalar function NVL returns the value of the first expression unless it evaluates to NULL. If the first expression evaluates to NULL, NVL returns the value of the second expression. The result data type is the same as the first expression.

Format

NVL (expression1, expression2)

Argument	Description
expression1	(required) Value you want returned (if expression1 is not NULL) expressed as an identifier or a value. Any data type is valid.
expression2	(required) Value you want returned (if expression1 is NULL) expressed as an identifier or a value. The data type must be compatible with expression1.

Example

The following SELECT statement, using the scalar function NVL, computes and displays total salary by adding the value of commission (COMM) to each SALARY in the EMPLOYEE table. If there is no commission, the example adds zero to the SALARY.

```
SELECT SALARY + NVL (COMM, 0) 'TOTAL SALARY'
FROM EMPLOYEE
```

OCTET_LENGTH

The scalar function OCTET_LENGTH returns the length, in bytes, of an expression. The result data type is INTEGER. If the expression is NULL, the result is NULL.

Format

OCTET_LENGTH (expression)

Argument	Description
expression	(required) Expression to be searched. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.

Example

The following SELECT statement, using the scalar function OCTET_LENGTH, returns the length of the data in column NAME in octets (bytes).

```
SELECT OCTET_LENGTH(NAME) FROM CUSTOMER
```

OVERLAY

The scalar function OVERLAY replaces a substring from the first expression with the second expression. The result data type depends on the data type of the first expression.

If the first expression is	Then the result data type is
CHAR or VARCHAR	VARCHAR
BINARY or VARBINARY	VARBINARY
BLOB	BLOB
CLOB	CLOB

Format

OVERLAY (expression1 PLACING expression2 FROM start_position [FOR length])

Argument	Description
expression1	(required) Substring to be replaced. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.
expression2	(required) String to replace expression1. The data type must be compatible with expression1, that is, if expression1 is a CHAR then expression2 must be CHAR or VARCHAR. If expression1 is CLOB, then expression2 must be CHAR, VARCHAR, or CLOB. If expression1 is BLOB, then expression2 must be BINARY, VARBINARY, or BLOB.
start_position	(required) Starting position of the first character or byte to be replaced.
length	(optional) Length of the substring to be replaced. If you omit the length, expression2 replaces every character or byte in expression1 from the specified starting position to the end of the string.

Example

The following SELECT statement, using the scalar function OVERLAY, changes the area code to 858 in phone numbers with a 619 area code.

```
SELECT NAME, OVERLAY(PHONE PLACING '858' FROM 1)
FROM CUSTOMER
WHERE SUBSTR(PHONE,1,3) = '619';
```

POSITION

The scalar function POSITION determines the starting position at which the first expression is found in the second expression. The result data type is INTEGER. The POSITION function is similar to the INSTR function, except you can specify a starting position and an occurrence with the INSTR function.

Format

POSITION (expression1 IN expression2)

Argument	Description
expression1	(required) Expression to be located in expression2. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.
expression2	(required) Expression to be searched. The data type must be compatible with expression1, that is, if expression1 is a CHAR (character string) then expression2 must be CHAR or VARCHAR. If expression1 is CLOB, then expression2 must be CHAR, VARCHAR, or CLOB. If expression1 is BLOB, then expression2 must be BINARY, VARBINARY, or BLOB.

Example

The following SELECT statement, using the scalar function POSITION, selects the customer number and name from all the rows in the CUSTOMER table where the string 'heritage' appears somewhere in the address column.

```
SELECT CUST_NO, NAME
FROM CUSTOMER
WHERE POSITION('heritage' IN ADDR) > 0
```

QUARTER

The scalar function QUARTER evaluates the specified date expression to determine the quarter of the year in which the date occurred. It returns the quarter of the year as a SMALLINT ranging from 1 to 4. If the date expression evaluates to NULL, the result is NULL.

Format

QUARTER (date_expression)

Argument	Description
date_expression	(required) Date for which you want the quarter returned expressed as an identifier or a value. The data type must be DATE.

Example

The following SELECT statement, using the scalar function QUARTER, selects all rows from the ORDERS table where the order date occurred in the third quarter.

```
SELECT *
FROM ORDERS
WHERE QUARTER (ORDER_DATE) = 3
```

RPAD

The scalar function RPAD pads to the right of the character expression with the characters specified in the padding expression. The second argument, length, specifies the length of the resulting character string. The result data type is the same as the character expression, that is, CHAR, CLOB, or VARCHAR. If the first argument evaluates to NULL, the result is NULL.

RPAD works as follows:

If the character expression is	Then
= to the specified length	No padding occurs
> than the specified length	The expression is truncated to the specified length

Format

RPAD (char_expression, length [, pad_expression])

Argument	Description
char_expression	(required) String to be padded to the right expressed as an identifier or a value. The data type must be CHAR, CLOB, or VARCHAR.
length	(required) Resulting length of the char_expression after padding. The data type must be INTEGER.
pad_expression	(optional) Character string used to pad char_expression. If specified, the data type must be CHAR. If omitted, the character string is padded with blanks.

Example

The following SELECT statement, using the scalar function RPAD, pads the names in the CUSTOMER table to the right with periods until each NAME is 30 characters long.

```
SELECT RPAD (NAME, 30, '.')
FROM CUSTOMER
```

RTRIM

The scalar function RTRIM removes all specified characters or bytes from the end of an expression. The result data type is the same as the expression type. If the expression evaluates to NULL, the result is NULL. If the expression is a CHAR data type, trailing blanks are preserved. If the result has a zero length, then a VARCHAR value with a zero length is returned.

Format

RTRIM (expression, char_set)

Argument	Description
expression	(required) Expression whose trailing characters or bytes will be deleted, expressed as an identifier or a value. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.
char_set	(required) One or more single characters or bytes, any of which are removed if they are the last characters or bytes in expression. If the expression is a character type (CHAR, VARCHAR, CLOB) then the char_set must be of character type (including a character literal). If the expression is a binary type (BINARY, VARBINARY, BLOB) then the char_set must be binary (including a hex literal). The default char_set is ' ' (blank) for character types and x'00' for binary types.

Example

The following SELECT statement, using the scalar function RTRIM, deletes all trailing spaces from the addresses (ADDR) in the CUSTOMER table and then pads the addresses to the right with periods until they are 30 characters long.

```
SELECT RPAD (RTRIM (ADDR, ' '), 30, '.')
FROM CUSTOMER
```

SECOND

The scalar function SECOND returns the number of seconds in the argument as a SMALLINT ranging from 0 to 62. If the time expression evaluates to NULL, the result is NULL.

Format

SECOND (time_expression)

Argument	Expression
time_expression	(required) Time field whose seconds will be converted to an integer expressed as an identifier or a value. The data type must be TIME or TIMESTAMP.

Example

The following SELECT statement, using the scalar function SECOND, selects all rows from the ARRIVALS table where the seconds portion of the IN_TIME is less than or equal to 40 seconds.

```
SELECT *
FROM ARRIVALS
WHERE SECOND (IN_TIME) <= 40
```

SUBSTR

The scalar function SUBSTR returns a substring of an argument based on a starting position in a specified expression and a length. The result data type is the same as the expression, that is, BLOB, CHAR, CLOB, or VARCHAR. If the argument evaluates to NULL, the result is NULL.

Format

SUBSTR (expression, start_position [, length])
or
SUBSTR (expression FROM start_position [FOR length])

Argument	Description
expression	(required) String (expressed as an identifier or a value) used to create the substring. The data type must be BLOB, CHAR, CLOB, or VARCHAR.
start_position	(required) Position in the expression where the substring begins, where 1 is the first position, 2 is the second position, and so on. The data type must be INTEGER.
length	(optional) Length of the substring. If specified, the data type must be INTEGER. If you omit the length, the substring contains every character or byte in the expression from the specified starting position to the end of the string.

Example

The following SELECT statement, using the scalar function SUBSTR, selects names from the CUSTOMER table and displays the corresponding phone number as three substrings in the format (999)999-9999.

```
SELECT NAME, '(', SUBSTR (PHONE, 1, 3) , ')',
SUBSTR (PHONE, 4, 3), '-',
SUBSTR (PHONE, 7, 4)
FROM CUSTOMER
```

or

```
SELECT NAME, '(', SUBSTR (PHONE FROM 1 FOR 3) , ')',
SUBSTR (PHONE FROM 4 FOR 3), '-',
SUBSTR (PHONE FROM 7 FOR 4)
FROM CUSTOMER
```

SUM

The aggregate function SUM returns the sum of the values in a specified group. A NULL result is permitted. The result data type is as follows:

If the input type is	Then the result type is
REAL	DOUBLE
DOUBLE	DOUBLE
SMALLINT	DECIMAL(15)
INTEGER	DECIMAL(20)
DECIMAL(p1,s1)	p = MIN (31, p1 + 10) s = s1

Format

SUM ({[ALL] expression} | {DISTINCT column_ref})

Argument	Description
ALL	(optional) Include duplicate values in the calculation of the sum.
DISTINCT	(optional) Exclude duplicate values in the calculation before computing the sum.
expression	(required if you omit column_ref) Group of identifiers or values whose sum will be calculated. The data type must be SMALLINT, INTEGER, DECIMAL, NUMERIC, REAL or DOUBLE PRECISION.
column_ref	(required if you omit expression) Name of the column whose values will be summed. The data type must be SMALLINT, INTEGER, DECIMAL, NUMERIC, REAL or DOUBLE PRECISION.

Example

The following SELECT statement, using the aggregate function SUM, calculates the sum of the AMOUNT column where the order date is the same as the current (system) date.

```
SELECT SUM (AMOUNT)
FROM ORDERS
WHERE ORDER_DATE = SYSDATE
```

TO_CHAR

The scalar function TO_CHAR converts a specified expression to character form and returns the result. The function uses the second argument, if specified, for conversion. The result data type is VARCHAR if the expression is CLOB or CHAR if the expression is any other data type except BLOB, which is not a supported conversion. If any arguments evaluate to NULL, the result is NULL.

Format

TO_CHAR (expression [,format])

Argument	Description
expression	(required) String to be converted to characters expressed as an identifier of a value. Any data type except BLOB is valid.
format	(optional) Format of the resulting character string. If specified, the data type must be CHAR. Currently, you can specify format only when the expression data type is DATE. See page 2-15 for valid date format strings.

Example

The following SELECT statement, using the scalar function TO_CHAR, selects names and start dates from the CUSTOMER table and converts all start dates to the format DD-MON-YYYY.

```
SELECT NAME, TO_CHAR (START_DATE, 'DD-MON-YYYY')
FROM CUSTOMER
```

TO_DATE

The scalar function TO_DATE converts a specified character expression to a date value. The second argument, if specified, supplies the conversion format. The result data type is DATE. If any argument evaluates to NULL, the result is NULL.

Format

TO_DATE (char_expression [,format])

Argument	Description
char_expression	(required) Character string to be converted to a date expressed as an identifier or a value. The data type must be CHAR, CLOB, or VARCHAR.
format	(optional) Format for the date. If specified, the data type must be CHAR. If omitted, the default format is MM/DD/YYYY, which is currently the only format supported.

Example

The following SELECT statement, using the scalar function TO_DATE, selects all rows from the ORDERS table where the order date is less than or equal to 12/31/1991.

```
SELECT *
FROM ORDERS
WHERE ORDER_DATE <= TO_DATE ('12/31/1991')
```

All dates are displayed in the format MM/DD/YYYY.

TO_HEX

The scalar function TO_HEX lets you compare binary columns with a character string. This function in combination with the LIKE function facilitates wildcard searches on binary data. TO_HEX converts each byte in the specified binary column to two characters. For each converted byte, the first character is the hexit representation of the first four bits, and the second character is the hexit representation of the last four bits. The result type is CHAR if the expression is BINARY or VARBINARY or CLOB if the expression is a BLOB. If the column evaluates to NULL, the result is NULL.

Format

TO_HEX (expression)

Argument	Description
expression	(required) Name of the column to be converted to a character string. The data type must be BINARY, BLOB, or VARBINARY.

Examples

- The following SELECT statement, using the scalar function TO_HEX, selects an entire row from the TABLE1 table for all telephone numbers that begin with area code 703. The WHERE clause with the LIKE function indicates a wildcard search.

  ```
  SELECT *
  FROM TABLE1
  WHERE TO_HEX (PHONE) LIKE '703%'
  ```

  In this example, PHONE is a BINARY column with two telephone number digits stored in each byte. TO_HEX converts each byte in PHONE to two characters for comparison with the character string '703'.

- The following SELECT statement, using the scalar function TO_HEX, selects all telephone numbers from TABLE1. The example assumes that telephone numbers are stored as binary data types. TO_HEX converts those binary formats to character strings for display purposes.

  ```
  SELECT TO_HEX(PHONE)
  FROM TABLE1
  ```

TO_NUMBER

The scalar function TO_NUMBER converts a specified character expression to a number value. The TO_NUMBER function will return data with a precision of 31 and a scale of 8. The result data type is NUMERIC. If any argument evaluates to NULL, the result is NULL.

Format

TO_NUMBER (char_expression)

Argument	Description
char_expression	(required) Character string to be converted to a number expressed as an identifier or a value. The data type must be CHAR, CLOB, or VARCHAR.

Example

The following SELECT statement, using the scalar function TO_NUMBER, selects all rows from the CUSTOMER table where the first three digits, or area code, of the telephone number equal 603.

```
SELECT *
FROM CUSTOMER
WHERE TO_NUMBER (SUBSTR (PHONE, 1, 3)) = 603
```

TO_TIME

The scalar function TO_TIME converts a specified character expression to a time value. The second argument, if specified, supplies the conversion format. The result data type is TIME. If any argument evaluates to NULL, the result is NULL.

Format

TO_TIME (char_expression [,format])

Argument	Description
char_expression	(required) Character string to be converted to a time expressed as an identifier or a value. The data type must be CHAR, CLOB, or VARCHAR.
format	(optional) Format for the time. If specified, the data type must be CHAR. See page 2-16 for valid time format strings. If omitted, the default format is HH:MM:SS.CCC.

Example

The following SELECT statement, using the scalar function TO_TIME, selects all rows from the ORDERS table where the order date is less than 05/15/1991 and the time of the order is less than 12 noon.

```
SELECT *
FROM ORDERS
WHERE ORDER_DATE < TO_DATE ('05/15/1991')
AND ORDER_TIME < TO_TIME ('12:00:00')
```

TRANSLATE

The scalar function TRANSLATE translates each character in char_expression that is contained in from_set, to the corresponding character in to_set. The translated string is the same as the expression, that is, data type CHAR, CLOB, or VARCHAR. If any argument evaluates to NULL, the result is NULL.

Format

TRANSLATE (char_expression, from_set, to_set)

Argument	Description
char_expression	(required) Character expression to be translated. The data type must be CHAR, CLOB, or VARCHAR.
from_set	(required) Characters in char_expression to be translated. The data type must be CHAR, CLOB, or VARCHAR.
to_set	(required) Replacement characters for the characters in from_set. The data type must be CHAR, CLOB, or VARCHAR.

For each character in from_set, there should be a corresponding character in to_set. In other words, from_set and to_set are usually the same length.

If to_set is *n* characters longer than from_set, then only *n* leading characters from to_set are considered (for example, if to_set is five characters and from_set is three characters, only the first three characters are replaced).

If to_set is shorter than from_set, no translation is performed for the trailing characters in from_set that do not have a corresponding character in to_set (for example, if to_set is five characters and from_set is seven characters, only the first five characters are replaced).

Example

The following SELECT statement, using the scalar function TRANSLATE, selects information in the NAME column from the CUSTOMER table and replaces all blanks in the street name with underscores.

```
SELECT NAME, TRANSLATE (STREET, ' ', '_')
FROM CUSTOMER
```

TRIM

The scalar function TRIM performs the same operations as LTRIM (removes leading characters or bytes) and RTRIM (removes trailing characters or bytes) but can also remove both leading and trailing characters or bytes in one operation. The default is to trim both leading and trailing characters or bytes. The default trim character is blank for CLOB, CHAR, and VARCHAR and X'00' for BLOB, BINARY, and VARBINARY. The result data type depends on the data type of the first expression.

If the first expression is	Then the result data type is
CHAR or VARCHAR	VARCHAR
BINARY or VARBINARY	VARBINARY
BLOB	BLOB
CLOB	CLOB

Format

TRIM ([[LEADING | TRAILING | BOTH] [expression1] FROM] expression2)

Argument	Description
LEADING	(optional) Remove the specified characters or bytes (expression1) from the beginning of expression2. TRIM (LEADING) is the same as the LTRIM function.
TRAILING	(optional) Remove the specified characters or bytes (expression1) from the end of expression2. TRIM (TRAILING) is the same as the RTRIM function.
BOTH	(optional) Remove both leading and trailing characters or bytes (expression1) from expression2. TRIM (BOTH) is the same as LTRIM followed by RTRIM.

Argument	Description
expression1	(optional) One or more characters or bytes to trim. The data type must not be BLOB or CLOB. If you omit expression1, the default trim character is blank for character types and X'00' for binary types.
expression2	(required) Expression to be trimmed. The data type must be BINARY, BLOB, CHAR, CLOB, VARBINARY, or VARCHAR.

Example

The following SELECT statement, using the scalar function TRIM, selects names and addresses from the CUSTOMER table and deletes all leading and trailing spaces from the address.

```
SELECT NAME, TRIM(BOTH ' ' FROM ADDR)
FROM CUSTOMER
```

UPPER

The scalar function UPPER converts the specified character expression to uppercase. The result data type is the same as the expression, that is, CHAR, CLOB, or VARCHAR. If the character expression evaluates to NULL, the result is NULL.

Format

UPPER (char_expression)

Argument	Description
char_expression	(required) Expression to be converted to uppercase. The data type must be CHAR, CLOB, or VARCHAR.

Example

The following SELECT statement, using the scalar function UPPER, converts the NAME column values in the CUSTOMER table to uppercase for comparison with the value SMITH.

```
SELECT *
FROM CUSTOMER
WHERE UPPER (NAME) = 'SMITH'
```

WEEK

The scalar function WEEK returns the week of the year as a SMALLINT ranging from 1 to 53. If the date expression evaluates to NULL, the result is NULL.

Format

WEEK (date_expression)

Argument	Description
date_expression	(required) Date of the year to be expressed as week of the year. The data type must be DATE.

Example

The following SELECT statement, using the scalar function WEEK, selects all rows in the ORDERS table where the ORDER_DATE occurred in the fifth week of the year.

```
SELECT *
FROM ORDERS
WHERE WEEK (ORDER_DATE) = 5
```

YEAR

The scalar function YEAR returns the year as a SMALLINT ranging from 0 to 9999. If the date expression evaluates to NULL, the result is NULL.

Format

YEAR (date_expression)

Argument	Description
date_expression	(required) Date to be expressed as the year. The data type must be DATE.

Example

The following SELECT statement, using the scalar function YEAR, selects all rows from the ORDERS table where the year portion of the ORDER_DATE equals 1992.

```
SELECT *
FROM ORDERS
WHERE YEAR (ORDER_DATE) = 1992
```

Appendix

A

SQL status codes

This appendix contains status codes generated in response to your StorHouse SQL statements. These codes are in addition to any codes your standard interface may generate.

Note: StorHouse/RM also returns StorHouse status codes in SQL status codes. Six-digit codes between -300000 and -309999 indicate StorHouse errors. Ignore the -30 prefix to obtain the 4-digit StorHouse status code. For example, SQL status code -304458 translates to StorHouse status code 4458. Refer to the StorHouse *Messages and Codes Manual* for more information about StorHouse status codes.

List of codes

0	Status OK
100	No rows or no more rows
-10000	SQL internal error - field exists
-10001	SQL internal error - field not found
-10002	Tuple not found
-10003	SQL internal error - end of tuple
-10004	SQL internal error - invalid hint

-10005	Segment ID mismatch
-10006	Table ID mismatch
-10007	Not enough overflow pages
-10008	SQL internal error - SegID null in fetch
-10009	SQL internal error - PrevtID not found
-10010	SQL internal error - not enough space
-10011	SQL internal error - base page mismatch
-10100	Index/sort key too long
-10101	SQL internal error - index scan end
-10102	Duplicate key
-10103	SQL internal error - index not found
-10104	Key desc field too long
-10105	Cursor inconsistent
-10106	Index method not supported
-10200	SQL internal error - PTPL overflow
-10201	First file size is small
-10300	SQL internal error - invalid lock type
-10301	Global lock table in invalid state
-10302	Lock wait error

-10303 Lock table exceeded maximum number of entries

-10304 Application deadlock

-10305 Lock upgrade failed

-10306 Hit 100-table-lock limit for a DB

-10307 Hit 100-lock limit for a table

-10311 Internal shared memory error

-10312 Shared memory permission denied

-10315 Out of system shared memory resource

-10316 Shared memory already used

-10317 Shared memory does not exist error

-10321 Internal semaphore error

-10322 Semaphore permission denied

-10323 Semaphore already removed error

-10324 Semaphore interrupt error

-10325 Out of system semaphore resource

-10326 Semaphore already used

-10327 Semaphore does not exist error

-10400 Bad log file size

-10401 Cannot find log file

-10402 Error in opening log file

-10403 Log file busy

-10404 Bad log entry

-10405 SQL internal error - log file not found

-10406 SQL internal error - end of log file

-10407 Database not recovered properly

-10408 Not enough log space

-10409 Unable to open log file

-10410 Unable to read log file

-10411 Log file sizes not equal

-10412 Log name too long

-10413 Invalid log entry during snapshot rollback

-10414 Wrong log file during roll forward

-10415 Failure writing archive file

-10499 SQL internal error - checkpoint

-10600 SQL internal error - in tablespace init

-10601 Cannot create any more tablespaces

-10602 Bad file for tablespace

-10603 SQL internal error - end of tablespace

-10604	SQL internal error - invalid tablespace number
-10605	Cannot add any more files to tablespace
-10606	SQL internal error - invalid hint
-10607	Page allocation failure in tablespace
-10608	Invalid database file size
-10609	Invalid tablespace number
-10700	SQL internal error - file number too large
-10701	File open error
-10800	SQL internal error - invalid dsl number
-10801	SQL internal error - dsl deadlock
-10900	SQL internal error - event timeout
-10920	Duplicate value in sorted list
-11100	Invalid transaction ID
-11101	No more resources for starting a transaction
-11102	Transaction space allocation failure
-11103	SQL internal error - tds not found
-11104	Transaction marked for abort
-11105	Transaction already active
-11106	Co-ordinator not set for the transaction

-11107	Cannot start new transactions - transactions disabled
-11108	Invalid consistency level
-11109	Invalid transaction handle
-11110	Transaction entry locked
-11111	Bad isolation level
-11199	SQL internal status code - transaction ended
-11200	Old snapshot
-11201	Invalid snapshot LSN
-20000	SQL internal error
-20001	Memory allocation failure
-20002	Open database failed
-20003	Syntax error
-20004	User not found
-20005	Table/view/synonym not found
-20006	Column not found/specified
-20007	No columns in table
-20008	Inconsistent types
-20009	Column ambiguously specified
-20010	Duplicate column specification

-20011 Invalid length

-20012 Invalid precision

-20013 Invalid scale

-20014 Missing input parameters

-20015 Subquery returns multiple rows

-20016 Null value supplied for a mandatory (NOT NULL) column

-20017 Too many values specified

-20018 Too few values specified

-20019 Cannot modify table referred to in subquery

-20020 Bad column specification for GROUP BY clause

-20021 Non-group-by expression in HAVING clause

-20022 Non-group-by expression in SELECT clause

-20023 Aggregate function not allowed here

-20024 Sorry, operation not implemented yet

-20025 Nested aggregate functions

-20026 Too many table references

-20027 Bad field specification in ORDER BY clause

-20028 An index already exists with same name

-20029 Index referenced not found

-20030 Tablespace with same name already exists

-20031 Cluster with same name already exists

-20032 No cluster with this name

-20033 Tablespace not found

-20034 Bad free percentage specification

-20035 Either a column specification or the NULL clause should be specified

-20036 Statement not prepared

-20037 Executing SELECT statement

-20038 Cursor not closed

-20039 Open for non-SELECT statement

-20040 Cursor not opened

-20041 Table/View/Synonym already exists

-20042 DISTINCT specified more than once in query

-20043 Tuple size too high

-20044 Array size too high

-20045 File does not exist or is not accessible

-20046 Field value NOT NULL for some tuples

-20047 Granting to self not allowed

-20048 Revoking for self not allowed

-20049 Keyword used for a name

-20050 Too many fields specified

-20051 Too many indexes on this table

-20052 Overflow error

-20053 Database not opened

-20054 Database not specified or improperly specified

-20055 Database not started

-20056 No DBA access rights

-20057 No RESOURCE privileges

-20058 Executing SQL statement for an aborted transaction

-20059 No files in the tablespace

-20060 Table not empty

-20061 Input parameter size too high

-20062 Full pathname not specified

-20063 Duplicate file specification

-20064 Invalid attach type

-20065 Invalid statement type

-20066 Invalid SQLDA

-20067 More than one database can't be attached locally

-20068 Bad arguments

-20069 SQLDA size not enough

-20070 SQLDA buffer length too high

-20071 Specified operation not allowed on the view

-20072 Server is not allocated

-20073 Query specification for view too long

-20074 View column list must be specified as expressions are given

-20075 Number of columns in column list is less than in SELECT list

-20076 Number of columns in column list is more than in SELECT list

-20077 Check option specified for non-insertable view

-20078 Given SQL statement is not allowed on the view

-20079 More tables cannot be created

-20080 View CHECK option violation

-20081 Number of expressions projected on either side of set-op don't match

-20082 Column names not allowed in ORDER BY clause for this statement

-20083 Outer join specified on a complex predicate

-20084 Outer join specified on a subquery

-20085 Invalid outer join specification

-20086 Duplicate table constraint specification

-20087 Column count mismatch

-20088 Invalid user name

-20089 System date retrieval failed

-20090 Table column list must be specified as expressions are given

-20091 Query statement too long

-20092 No tuples selected by the subquery for update

-20093 Synonym already exists

-20094 Database link with same name already exists

-20095 Database link not found

-20096 Connect string not specified or specified incorrectly

-20097 Specified operation not allowed on a remote table

-20098 More than one row selected by the query

-20099 Cursor not positioned on a valid row

-20100 Subquery not allowed here

-20101 No references for the table

-20102 PRIMARY/CANDIDATE key column defined NULL

-20103 No matching key defined for the referenced table

-20104 Keys in reference constraint incompatible

-20105 Statement not allowed in read-only isolation level

-20106 Invalid ROWID

-20107 Remote database not started

-20108 Remote network server not started

-20109 Remote database name not valid

-20110 TCP/IP remote hostname is unknown

-20111 Target of DROP TABLE is not a table

-20112 Target of DROP VIEW is not a view

-20113 Fetched value truncated and no indicator var

-20114 Fetched value NULL and no indicator var

-20115 References to the table/record present

-20116 Constraint violation

-20117 Table definition not complete

-20118 Duplicate constraint name

-20119 Constraint name not found

-20120 Use of reserved word

-20121 Permission denied

-20122 Procedure not found

-20123 Invalid arguments to procedure

-20124 Tablespace not empty

-20125 Tablespace not specified and there is no default

-20126 The size of the result set is over the limit

-20127 Query conditionally terminated

-20128 Number of open cursors exceeds limit

-20129 Invalid cursor name

-20130 Bad parameter specification for the statement

-20131 DELETE/INSERT/UPDATE to user table not allowed

-20132 SQLDA sqld_length not an aligned value for sqld_type

-20133 Table privileges cannot be revoked from the table owner

-20134 Failed to configure system soft limit on number of file descriptors

-20135 SQL internal error - no filename for segment

-20136 Invalid database name syntax

-20137 Invalid database directory

-20138 Database name has already been used

-20139 Database name is misspelled

-20140 The subspace does not exist

-20141 Use ALTER TABLESPACE with SUBSPACE clause(s) for this table space

-20142 Pointer-fetch mode not allowed

-20143 Constraint is only applicable to LOB datatypes

-20144 LOB STORE WITH column not found/specified

-20145 Invalid expression for No-table SELECT (or VALUES INTO)

-20146 LOB_FILE not allowed with array fetch

-20147 LOB datatypes are not allowed on the DISTINCT, ORDER BY or GROUP clauses

-20148 Illegal view operation. Insert/update/delete not allowed

-20149 MAX EXTENT limit of 32767 exceeded

-20160 Illegal ON CLAUSE table reference

-20211 Remote procedure call error

-20212 SQL client bind to daemon failed

-20213 SQL client bind to SQL server failed

-20214 SQL NETWORK service entry is not available

-20215 Invalid TCP/IP hostname

-20216 Invalid remote database name

-20217 Network error on server

-20218 Invalid protocol

-20219 Invalid connection name

-20220 Duplicate connection name

-20221 No active connection

-20222 No environment-defined database

-20223 Multiple default (DB_NAME used) connections

-20224 Invalid protocol in connect_string

-20225 Exceeding permissible number of connections

-20226 Bad database handle

-20227 Invalid host name in connect string

-20228 Access denied (authorization failed)

-20229 Invalid date value

-20230 Invalid date string

-20231 Invalid number value

-20232 Invalid number string

-20233 Invalid time value

-20234 Invalid time string

-20235 Invalid timestamp string

-20236 Division by zero attempted

-20237 Invalid hexit string

-20238 Database offline and unavailable for access

-20240 LOB_FILE not enabled for writing

-20241 LOB_FILE not enabled for reading

-20242 LOB_FILE specifies TPE_FILE_CREATE, but file exists

-20243	Error opening LOB_FILE
-20244	Error writing LOB_FILE
-20245	Invalid Locator
-20246	Disjoint Modify of lob
-20247	Join columns may not contain lob data
-20248	Unique qualifier not allowed on user table indexes
-20249	Storage_manager table space mismatch
-20300	Column group column doesn't exist
-20301	Column group column already specified
-20302	Column group name already specified
-20303	Column groups haven't covered all columns
-20304	Column groups are not implemented in storage
-30001	Query aborted on user request
-30002	Invalid network handle
-30003	Invalid SQL network interface
-30004	Invalid SQL network interface procedure
-30005	Interface is already attached
-30006	Interface entry not found
-30007	Interface is already registered

-30008 Mismatch in pkt header size and total argument size

-30009 Invalid server ID

-30010 Reply does not match the request

-30011 Memory allocation failure

-30012 Communication packet overflow failure

-30013 Output SQLDA changed between FETCH requests

-30031 Error in transmission of packet to server

-30032 Error in reception of packet from server

-30033 Server unexpectedly lost

-30034 Server unexpectedly lost (ECONNRESET on send)

-30035 Server unexpectedly lost (ECONNRESET on recv)

-30041 Error in transmission of packet to client

-30042 Error in reception of packet from client

-30043 Client unexpectedly lost

-30051 Network handle is in-process handle

-30061 Could not connect to SQL network daemon

-30062 Error in number of arguments

-30063 Requested interface not registered

-30064 Invalid interface procedure ID

-30065	Requested server executable not found
-30066	Invalid configuration information
-30067	Interface not supported
-30091	Invalid service name
-30092	Invalid host
-30093	Error in TCP/IP accept call
-30094	Error in TCP/IP connect call
-30095	Error in TCP/IP bind call
-30096	Error in creating socket
-30097	Error in setting socket option
-30101	Interrupt occurred
-30201	Error setting up FTP connection to host
-30202	Couldn't ftp login to remote host
-30203	Internal error - no FTP connection exists
-30204	Error initiating ftp get command or file not found
-30205	Error reading LOB file from host
-30206	Error initiating ftp put command
-30207	Error writing LOB file to host
-40001	Error in reading configuration

-50001 Flat File: File IO error

-50002 Flat File: No more records

-50003 Flat File: Table already exists

-50004 Flat File: Invalid record number

-50005 Flat File: Record already deleted

-50007 Flat File: Inserting duplicate value into unique index

-50008 Flat File: Illegal attempt to update system table entry

-50009 Flat File: Illegal attempt to delete system table entry

-50010 Flat File: Permission denied on creating directory

-50011 Flat File: Directory already exists

-50012 Flat File: Create database failed

-50013 Flat File: Database does not exist

-50014 Flat File: Permission denied on changing directory

-50015 Flat File: Fail to open database

-50016 Flat File: Open CONFIG.CGF failed

-50017 Flat File: Error writing control record

-50018 Flat File: Error reading control record

-50019 Flat File: Error opening file

-50020 Flat File: Error writing record

-50021 Flat File: Error reading record

-50022 Flat File: Error creating table/index log file

-50023 Flat File: Error opening table/index log file

-50024 Flat File: Error deleting table/index log file

-50025 Flat File: Error writing table/index log file

-50026 Flat File: Error reading table/index log file

-50027 Flat File: Error reading CONFIG.CGF

-50028 The total length (bytes) of a key is too long.

Note: The total length of a key includes the byte-length of all fields (single or compound) and encode information inserted for each field. The maximum key length is 256 bytes.

-50029 Fail to insert a key on index file.

-50030 Fail to delete a key from index file.

-50031 Error while searching a key on index file.

-50032 Fail to create a new index file.

-50033 Fail to open an existing index file.

-50036 Error creating DB alias file

-50037 Error opening DB alias file

-50038 Error reading DB alias file

-50039 Index file may be corrupt, should be recreated

-50040 Invalid flat file table space id

-50100 Flat File: create file error, no such directory

-50101 Flat File: create file error, I/O error

-50102 Flat File: create file error, resource not available

-50103 Flat File: create file error, permission denied

-50104 Flat File: create file error, file already exists

-50105 Flat File: create file error, file name is a directory name

-50106 Flat File: create file error, file table overflow

-50107 Flat File: create file error, no space available

-50108 Flat File: create file error, read only file system

-50109 Flat File: create file error, unexpected error

-50200 Flat File: open file error, no such file or directory

-50201 Flat File: open file error, I/O error

-50202 Flat File: open file error, resource not available

-50203 Flat File: open file error, permission denied

-50204 Flat File: open file error, file name is a directory name

-50205 Flat File: open file error, too many open files

-50206 Flat File: open file error, unexpected error

-50300 Flat File: seek file error

-50400	Flat File: write file error, I/O error
-50401	Flat File: write file error, bad file number
-50402	Flat File: write file error, resource not available
-50403	Flat File: write file error, no more space available
-50404	Flat File: write file error, unexpected error
-50500	Flat File: read file error, I/O error
-50501	Flat File: read file error, bad file number
-50502	Flat File: read file error, unexpected error
-50600	Flat File: close file error, bad file number
-50601	Flat File: close file error, unexpected error
-50700	Wrong system table version
-60001	Loader: Ran out of memory
-60002	Loader: Field mismatch
-60003	Loader: Error reading checkpoint file
-60004	Loader: Error closing checkpoint file
-60005	Loader: Data type not supported yet
-60006	Loader: Error initializing shared memory
-60010	Loader: Error writing checkpoint file
-60015	Loader: Error creating checkpoint file

-60016 Loader: Error opening checkpoint file

-60017 Loader: Data received past previous EOF

-60018 Loader: Invalid (old) checkpoint log file

-60019 Loader: Must use same inputs on restart

-60020 Loader: Table definition has changed

-60021 Loader: Unrecoverable segment file

-60022 Loader: Can't abort. Load already finished

-60023 Loader: Multiple SEGMENT tags for same segment

-60024 Loader: Multiple REPLACE tags for same segment

-60025 Loader: Can't use SUBSPACE n with SUBSPACE ROTATE

-60026 Loader: FIELDS CHAR not allowed with fields_spec list

-60027 Loader: FIELDS NULLFLAGS not allowed with field_spec list

-60028 Loader: LOB type fields must be last in fields_spec list

-60029 Loader: LOB type fields must be at POSITION(*)

-60030 Loader: LOB type field can only be used with a LOB column

-60031 Loader: Last LOB type field did not end at end of a record

-60032 Loader: LOB field data cannot be used in a comparison

-60200 Unloader: #expr mismatch between USING and SELECT

-60201 Unloader: No matching :field spec found

-60202 Unloader: No datatype specified for position_spec

-60203 Unloader: FIELDS CHAR not allowed with USING clause

-60204 Unloader: Output row exceeds maximum length

-60205 Unloader: Data too long for user field

-60206 Unloader: No FILENAME specified for LOB_FILE

-60207 Unloader: FIELDS NULLFLAGS not allowed with USING clause

-60208 Unloader: RECORDS terminator not allowed with LOB fields

-70000 Cannot use Extractor: Table is not a user table

-70003 Cannot use Extractor: Unsupported clauses in select statement

-70005 Cannot use Extractor: Incompatible SQLDA parameters

-90001 Internal Error: Error MMAPing file

-90002 Internal Error: Invalid array range or index

-90003 Internal Error: Buffer Overflow

-91001 Internal Error: (LM) Incompletely read LOB

-91002 Internal Error: (LM) LOB Cache Request too large

-91501 Internal Error: (XV) Invalid Offset

-95001 HS: Invalid table not initialized for operation

-95002 HS: LOB Scan object segID and/or subSegID mismatch

-95003 HS: OS write failed

-95004	HS: OS file space problem (possibly out of space)
-95201	TSM: Invalid parameter
-95202	TSM: OS file space problem (possibly out of space)
-95203	TSM: OS file write failed
-95204	TSM: OS file read failed
-95205	TSM: OS file seek failed
-95206	TSM: Max TSM allocated file space exceeded
-95207	TSM: Max TSM allocated files exceeded
-95208	TSM: Inconsistent file space count encountered
-95209	TSM: Inconsistent file count encountered
-95210	TSM: FDC file open failed
-95211	TSM: FDC get file descriptor failed
-200001	Unknown storage manager name
-300000–309999	Status codes between -300000 and -309999 indicate StorHouse errors. Ignore the -30 prefix to determine the 4-digit StorHouse status code and then refer to the StorHouse *Messages and Codes Manual* for more information.
-310200	Range index: System table for range index does not exist
-320100	Temp manager: Temporary directory undefined
-321001	Bad table type
-321002	User does not have SCAN privilege on the table

Appendix B

StorHouse SQL reserved words

This appendix lists the StorHouse SQL reserved words. In StorHouse, you can use reserved words as SQL identifiers (alias names, cursor names, and database component names) only if you delimit them with double quotes (“).

A
ABS
ADD
ADD_MONTHS
ALL
ALTER
AN
AND
ANY
ARRAY
AS
ASC
ASCII
AUTO
AVG
BEGIN
BETWEEN
BIGINT
BINARY
BIND
BINDING
BIT
BLOB
BY
CALL
CAST
CHAR
CHARACTER
CHECK
CHR
CLEANUP
CLOB
CLOSE
COLGROUP
COMMIT
COMPLEX
COMPRESS
CONNECT
CONNECTION
CONSTRAINT
CONTINUE
COUNT
CREATE
CURRENT
CURSOR
CVAR
DATABASE
DATE
DAYOFMONTH
DAYOFWEEK
DAYOFYEAR
DAYS
DBA
DEC
DECIMAL
DECLARATION
DECLARE
DECODE
DEFAULT
DEFINITION
DELETE
DESC
DESCRIBE
DESCRIPTOR

DISCONNECT
DISTINCT
DOUBLE
DROP
END
ESCAPE
EXCLUSIVE
EXEC
EXECUTE
EXISTS
EXIT
EXPLICIT
EXTERN
FETCH
FILE
FLOAT
FOR
FOREIGN
FOUND
FROM
GO
GOTO
GRANT
GREATEST
GROUP
HAVING
HOUR
IDENTIFIED
IMMEDIATE
IN
INDEX
INDEXPAGES
INDICATOR
INITCAP
INPUT
INSERT
INSTR
INT
INTEGER
INTERFACE
INTERSECT
INTO
IS
ISOLATION
JOIN
KEY
LARGE
LAST_DAY
LEAST
LEFT
LENGTH
LEVEL
LIKE
LINK
LIST
LOAD
LOCK
LOG
LONG
LOWER
LPAD
LTRIM
LVARBINARY
LVARCHAR
MAIN
MAX
MIN
MINUS
MINUTE
MODE
MODIFY
MONEY
MONTH
MONTHS_BETWEEN
NEXT_DAY
NOCOMPRESS
NOT
NOWAIT
NULL
NUMBER
NUMERIC
NVL
OBJECT
OF
ON
OPEN

OPTION
OR
ORDER
OUTER
OUTPUT
PCTFREE
PRECISION
PREPARE
PRIMARY
PRIVILEGES
PUBLIC
QUARTER
QUESTION
RAW
REAL
RECORD
REFERENCES
RENAME
RESOURCE
RETURN
REVOKE
ROLLBACK
ROWID
ROWNUM
RPAD
RTRIM
SECTION
SECOND
SELECT
SERVICE
SET
SHARE
SHORT
SIZE
SMALLINT
SOME
SPACE
SQL
SQLERROR
SQLWARNING
START
STATIC
STATISTICS
STOP
STORAGE_ATTRIBUTES
STORAGE_MANAGER
SUBSTR
SUM
SYNONYM
SYSDATE
SYSTIME
SYSTIMESTAMP
TABLE
TABLESPACE
TIME
TIMEOUT
TIMESTAMP
TINYINT
TO
TO_CHAR
TO_DATE
TO_NUMBER
TO_TIME
TPE
TRANSACTION
TRANSLATE
TYPE
UNION
UNIQUE
UNSIGNED
UPDATE
UPPER
USER
USING
UID
UUID
VALUES
VARBINARY
VARCHAR
VARIABLES
VERSION
VIEW
WEEK
WHENEVER
WHERE
WITH

WORK
YEAR

Appendix

C

Deprecated syntax

This appendix contains the deprecated syntax of ALTER TABLE SPACE, CREATE TABLE SPACE, and joins. StorHouse/RM releases 2.3 and higher accept the deprecated syntax, but convert it, so that your existing programs and scripts can continue to work.

ALTER TABLE SPACE

The ALTER TABLE SPACE statement for StorHouse/RM releases 2.2A and earlier consists of a USING clause with at least one parameter and value. With this format, you cannot change the GROUP value for the user tablespace.

Format

ALTER TABLE SPACE table_space_name
USING param_value[, ...]

where param_value is defined as:

TABLE_VSET vs_name | TABLE_FSET fs_name |
TABLE_ATF atf | TABLE_VTF vtf |
HASH_VSET vs_name | HASH_FSET fs_name |
HASH_ATF atf | HASH_VTF vtf |
VALUE_VSET vs_name | VALUE_FSET fs_name |
VALUE_ATF atf | VALUE_VTF vtf |
EDC edc

Argument	Description
table_space_name	(required) Name of the user tablespace to be altered.
param_value	(at least one required) Parameter(s) to be changed.
TABLE_VSET vs_name	Name of the volume set to contain the table data files for the user tablespace.
TABLE_FSET fs_name	Name of the file set to contain the table data files for this user tablespace.

Argument	Description
TABLE_ATF atf	Access Time Factor (ATF) for table data files in the user tablespace. Valid values are: ■ 0 – (default) Use the value of the StorHouse ATF system parameter. ■ 1 – Short access time is important. ■ 2 – Short access time is moderately important. ■ 3 – Short access time is minimally important.
TABLE_VTF vtf	Vulnerability Time Factor (VTF) for table data files in the user tablespace. Valid values are: ■ DEFAULT – (default) Use the value of the StorHouse VTF system parameter. ■ NOW – Write the file to the performance buffer first and then copy it immediately to its file set. ■ NEXT – Write the file to the performance buffer and copy it to its file set during the next StorHouse write-back operation. ■ DIRECT – Bypass the performance buffer and write the file directly to its file set.
HASH_VSET vs_name	Name of the volume set to contain hash index files for the user tablespace.
HASH_FSET fs_name	Name of the file set to contain hash index files for the user tablespace.
HASH_ATF atf	Access Time Factor (ATF) for hash index files in the user tablespace. Valid values are: ■ 0 – (default) Use the value of the StorHouse ATF system parameter. ■ 1 – Short access time is important. ■ 2 – Short access time is moderately important. ■ 3 – Short access time is minimally important.

Argument	Description
HASH_VTF vtf	Vulnerability Time Factor (VTF) for hash index files in the user tablespace. Valid values are: ■ DEFAULT – (default) Use the value of the StorHouse VTF system parameter. ■ NOW – Write the file to the performance buffer first and then copy it immediately to its file set. ■ NEXT – Write the file to the performance buffer and copy it to its file set during the next StorHouse write-back operation. ■ DIRECT – Bypass the performance buffer and write the file directly to its file set.
VALUE_VSET vs_name	Name of the volume set to contain value index files for the user tablespace.
VALUE_FSET fs_name	Name of the file set to contain value index files for the user tablespace.
VALUE_ATF atf	Access Time Factor (ATF) for value index files in the user tablespace. Valid values are: ■ 0 – (default) Use the value of the StorHouse ATF system parameter. ■ 1 – Short access time is important. ■ 2 – Short access time is moderately important. ■ 3 – Short access time is minimally important.

Argument	Description
VALUE_VTF vtf	Vulnerability Time Factor (VTF) for value index files in the user tablespace. Valid values are: ■ DEFAULT – (default) Use the value of the StorHouse VTF system parameter. ■ NOW – Write the file to the performance buffer first and then copy it immediately to its file set. ■ NEXT – Write the file to the performance buffer and copy it to its file set during the next StorHouse write-back operation. ■ DIRECT – Bypass the performance buffer and write the file directly to its file set.
EDC edc	Error Detection Code flag indicating whether the StorHouse error detection capability is to be used for table data and index files. Valid values are: ■ D – (default) Use the StorHouse system default for EDC. ■ Y – Use EDCs. ■ N – Do not use EDCs.

Example

The following ALTER TABLE SPACE statement changes the vulnerability time factor to DIRECT for hash index files in tablespace BILL98.

```
ALTER TABLE SPACE BILL98
USING HASH_VTF DIRECT
```

CREATE TABLE SPACE

The CREATE TABLE SPACE statement for StorHouse/RM releases 2.2A and earlier consists of a USING clause. Your existing programs and scripts can continue to use this old format, but if they do, StorHouse/RM releases 2.3 and higher convert the CREATE TABLE SPACE into three SUBSPACE clauses. See the examples on page C -9 for more information about this conversion. If you do not want three subspaces in your user tablespace, then change your programs or scripts to use the new CREATE TABLE SPACE format described on page 4-35 instead of the old format described below.

Format

CREATE TABLE SPACE table_space_name
USING TABLE_VSET vs_name, TABLE_FSET fs_name [,param_value...]

where param_value is any of the following:

TABLE_ATF atf | TABLE_VTF vtf |
HASH_VSET vs_name | HASH_FSET fs_name |
HASH_ATF atf | HASH_VTF vtf |
VALUE_VSET vs_name | VALUE_FSET fs_name |
VALUE_ATF atf | VALUE_VTF vtf |
EDC edc | GROUP group_name

Argument	Description
table_space_name	(required) Name of the new user tablespace. This name must be unique in the current database.
TABLE_VSET vs_name	(required) Name of the volume set to contain the table data files for the user tablespace.
TABLE_FSET fs_name	(required) Name of the file set to contain the table data files for the user tablespace.
param_value	(optional) List of optional storage parameters for the user tablespace.

Argument	Description
TABLE_ATF atf	(optional) Access Time Factor (ATF) for table data files in the user tablespace. Valid values are: ■ 0 – (default) Use the value of the StorHouse ATF system parameter. ■ 1 – Short access time is important. ■ 2 – Short access time is moderately important. ■ 3 – Short access time is minimally important.
TABLE_VTF vtf	(optional) Vulnerability Time Factor (VTF) for table data files in the user tablespace. Valid values are: ■ DEFAULT – (default) Use the value of the StorHouse VTF system parameter. ■ NOW – Write the file to the performance buffer first and then copy it immediately to its file set. ■ NEXT – Write the file to the performance buffer and copy it to its file set during the next StorHouse write-back operation. ■ DIRECT – Bypass the performance buffer and write the file directly to its file set.
HASH_VSET vs_name	(required if you specify HASH_FSET) Name of the volume set to contain hash index files for the user tablespace. The default value is the TABLE_VSET value.
HASH_FSET fs_name	(required if you specify HASH_VSET) Name of the file set to contain hash index files for the user tablespace. The default value is the TABLE_FSET value.
HASH_ATF atf	(optional) Access Time Factor (ATF) for hash index files in the user tablespace. Valid values are: ■ 0 – (default) ■ 1 – Short access time is important. ■ 2 – Short access time is moderately important. ■ 3 – Short access time is minimally important.

Argument	Description
HASH_VTF vtf	(optional) Vulnerability Time Factor (VTF) for hash index files in the user tablespace. Valid values are: ■ DEFAULT – (default) Use the value of the StorHouse VTF system parameter. ■ NOW – Write the file to the performance buffer first and then copy it immediately to its file set. ■ NEXT – Write the file to the performance buffer and copy it to its file set during the next StorHouse write-back operation. ■ DIRECT – Bypass the performance buffer and write the file directly to its file set.
VALUE_VSET vs_name	(required if you specify VALUE_FSET) Name of the volume set to contain value index files for the user tablespace. The default value is the TABLE_VSET value.
VALUE_FSET fs_name	(required if you specify VALUE_VSET) Name of the file set to contain value index files for the user tablespace. The default value is the TABLE_FSET value.
VALUE_ATF atf	(optional) Access Time Factor (ATF) for value index files in the user tablespace. Valid values are: ■ 0 – (default) Use the value of the StorHouse ATF system parameter. ■ 1 – Short access time is important. ■ 2 – Short access time is moderately important. ■ 3 – Short access time is minimally important.
VALUE_VTF vtf	(optional) Vulnerability Time Factor (VTF) for value index files in the user tablespace. Valid values are: ■ DEFAULT – (default) Use the value of the StorHouse VTF system parameter. ■ NOW – Write the file to the performance buffer first and then copy it immediately to its file set. ■ NEXT – Write the file to the performance buffer and copy it to its file set during the next StorHouse write-back operation. ■ DIRECT – Bypass the performance buffer and write the file directly to its file set.

Argument	Description
EDC edc	(optional) Error Detection Code flag indicating whether the StorHouse error detection capability is to be used for table data and index files. Valid values are: ■ D – (default) Use the StorHouse system default for EDC. ■ Y – Use EDCs. ■ N – Do not use EDCs.
GROUP group_name	(optional) Name of the StorHouse file access group to contain table data and index files. If you omit this parameter, the default value is STH.

Examples

■ The following CREATE TABLE SPACE statement creates a new user tablespace named BILL98. This example contains only the required arguments.

```
CREATE TABLE SPACE BILL98
USING TABLE_VSET ACCTVSET, TABLE_FSET ACCTFSET
```

StorHouse/RM releases 2.3 and higher convert this statement into three SUBSPACE clauses:

```
CREATE TABLE SPACE BILL98
( SUBSPACE 1 VSET ACCTVSET FSET ACCTFSET OBJECT_TYPE T,
SUBSPACE 2 VSET ACCTVSET FSET ACCTFSET OBJECT_TYPE H,
SUBSPACE 3 VSET ACCTVSET FSET ACCTFSET OBJECT_TYPE V )
```

In this example, hash and value index files are stored in the same volume set and file set as the table data files. The default values are used for the omitted parameters.

- The following CREATE TABLE SPACE statement creates a new user tablespace named BILLING1996. This example contains all of the required and optional arguments.

```
CREATE TABLE SPACE BILLING1996
USING
TABLE_VSET TABLE96,
TABLE_FSET MAY,
HASH_VSET HINDEX96,
HASH_FSET MAY,
VALUE_VSET VINDEX96,
VALUE_FSET MAY,
TABLE_VTF NEXT,
TABLE_ATF 0,
HASH_VTF NOW,
HASH_ATF 1,
VALUE_VTF NOW,
VALUE_ATF 1,
EDC Y,
GROUP B5
```

StorHouse/RM releases 2.3 and higher convert this statement into three SUBSPACE clauses:

```
CREATE TABLE SPACE BILLING1996
( SUBSPACE 1 VSET TABLE96 FSET MAY OBJECT_TYPE T ATF 0
VTF NEXT EDC Y GROUP B5 ,
SUBSPACE 2 VSET HINDEX96 FSET MAY OBJECT_TYPE H ATF 1
VTF NOW EDC Y GROUP B5 ,
SUBSPACE 3 VSET VINDEX96 FSET MAY OBJECT_TYPE V ATF 1
VTF NOW EDC Y GROUP B5 )
```

Joins

Before StorHouse/RM release 3.1, you specified a join operation by naming the tables on the FROM clause and by specifying the columns with their respective join conditions on the WHERE clause. You can continue to use the deprecated syntax but note the following considerations:

- For multiple outer-joins, particularly when combined with inner-joins, you should verify that the result sets are correct, that is, the join order matches the expected result.
- For multiple outer-joins, an error occurs if the table names are not in the correct order on the FROM clause. Specify the left-side table first for a left outer-join and the right-side table first for a right outer-join.
- If a query fails with SQL code -20085—Invalid outer join specification—then change the order of the table list in the FROM clause. If re-ordering the table list does not eliminate the error, then StorHouse/RM cannot support that outer-join structure.
- This syntax does not support full outer-joins (left and right outer-joins on the same columns).

Format

FROM table_name [correlation_name] [, table_name [correlation_name]]...
WHERE condition

Argument	Description
table_name	(required) Name of the table to be used in the join. For a left outer-join, specify the left table first. For a right outer-join, specify the right table first.
correlation_name	(optional for all join types except self-joins) Another name for the table to be joined with itself. Include a space between the table name and alias name, for example: CUSTOMER FIRST
condition	(required for all join types except Cartesian product) Condition used to select and/or combine rows. ■ If the column names are the same in both tables, qualify the column names with the table names. ■ For self-joins, qualify the column names with the alias names. ■ The format for a left outer-join condition is: WHERE [table_name.]column = [table_name.]column (+) ■ The format for a right outer-join condition is: WHERE [table_name.]column (+) = [table_name.]column

Examples

- The following left outer-join selects customers and their corresponding orders from two tables. The left table is CUSTOMERS and the right table is ORDERS. The SELECT statement returns all customers in the CUSTOMERS table, including those customers with no corresponding entry in the ORDERS table (in this case, order number and order date are returned with NULL values).

  ```
  SELECT CUSTOMERS.CUST_NO, CUSTOMERS.NAME,
  ORDERS.ORDER_NO, ORDERS.ORDER_DATE
  FROM CUSTOMERS, ORDERS
  WHERE CUSTOMERS.CUST_NO = ORDERS.CUST_NO (+)
  ```

- The following right outer-join preserves information from the table specified on the right in the WHERE clause, that is, all rows in the BILLSUMMARY table are selected, even if some rows do not have matching rows in the BILLDETAIL table. In the FROM clause, the right-side table precedes the left-side table. For example:

  ```
  FROM BILLSUMMARY,BILLDETAIL
  WHERE BILLDETAIL.ACCTNUM(+)=BILLSUMMARY.ACCTNUM
  ```

- The following equi-join selects the customer number, name, order number, and order date from the CUSTOMERS and ORDERS tables for all customer numbers that appear in both tables.

  ```
  SELECT CUSTOMERS.CUST_NO, CUSTOMERS.NAME,
  ORDERS.ORDER.NO, ORDERS.ORDER_DATE
  FROM CUSTOMERS, ORDERS
  WHERE CUSTOMERS.CUST_NO = ORDERS.CUST_NO
  ```

- The following self-join selects all customer names and numbers that are from the same city as customer SMITH from the CUSTOMER table. The two correlation names for the CUSTOMER table are FIRST and SECOND.

```
SELECT SECOND.CUST_NO, SECOND.NAME
FROM CUSTOMER FIRST, CUSTOMER SECOND
WHERE FIRST.NAME = ‘SMITH’ AND SECOND.CITY = FIRST.CITY
```

Index

Symbols

|| concatenation operator 2-23, 6-16

A

ABS scalar function 6-5

account ID 2-4

active set, definition 1-6

ADD_MONTHS scalar function 6-7

aggregate function, definition 6-1

aggregate functions, specific
- AVG 6-9
- COUNT 6-18
- MAX 6-37
- MIN 6-38
- SUM 6-55

alias names 2-5

ALL argument, SELECT statement 4-83

ALTER TABLE SPACE statement 4-7, C-2

AND boolean operator 5-5

AND logical operator 2-20

ANSI 1992 Level 2 SQL 1-1

answer set, definition 1-6

ANY keyword, quantified predicate 5-7

argument, definition 4-1

arithmetic operators 2-21

arrays, declaring 4-14

ASCII scalar function 6-8

auto-join, definition 1-8

AVG aggregate function 6-9

B

basic predicate 5-3

BEGIN DECLARE SECTION statement 4-11

BETWEEN predicate 5-9

BINARY data type 3-3

BIT_LENGTH scalar function 6-11

blanks in CHAR and VARCHAR comparisons 5-2

BLOB data type 3-4

BLOB scalar function 6-12

BLOB_FILE data type 3-22

BLOB_LOCATOR data type 3-22

boolean operator, definition 5-5

boolean operators, specific
- AND 2-20, 5-5
- NOT 2-20, 5-5
- OR 2-20, 5-5

BYTE data type 3-18

C

C language structures 3-21

C++-style comments 1-13

Cartesian product, definition 1-8

CHAR_LENGTH scalar function 6-13

CHARACTER data type 3-5

character literal, definition 2-11

CHR scalar function 6-14

clause, definition 2-2

clauses, specific
- DESCRIBE BIND VARIABLES FOR 4-46
- DESCRIBE SELECT LIST 4-47
- FOR 4-93
- FROM 4-86
- GROUP BY 4-89
- HAVING 4-90
- INTO (SELECT) 4-85
- INTO (VALUES) 4-108
- ORDER BY 4-91
- SET 4-106
- USING (EXECUTE) 4-57, 4-58
- USING (OPEN) 4-72
- USING DESCRIPTOR (EXECUTE) 4-58
- USING DESCRIPTOR (FETCH) 4-63
- USING DESCRIPTOR (OPEN) 4-72
- VALUES 4-71
- WHERE 4-88

CLOB data type 3-6

CLOB scalar function 6-15

CLOB_FILE data type 3-22

CLOB_LOCATOR data type 3-22

CLOSE statement 4-19

codes, error A-1

column constraints 4-32

comments
- C++-style 1-13
- C-style 1-13
- SQL-style 1-13

COMMIT WORK statement 4-20

comparison of CHAR and VARCHAR fields with blanks 5-2

comparison of data types in DB2 and StorHouse 3-16

comparison operator, definition 2-23

complex predicate 2-20, 5-5

component, definition 4-1

compound index, definition 4-25

CONCAT scalar function 6-16

concatenation operator, definition 2-23

concatenation result data types 2-24

CONNECT statement 4-22

connect string 4-22

connection name 4-22

constant 2-11

Control Center, description xx

conventions
- publication xxiii
- SQL examples 1-11
- SQL formats 1-10

conversion
- function data types 3-29
- literals 3-28
- loader data types 3-23
- unloader data type 3-26

COUNT aggregate function 6-18

CREATE INDEX statement 4-24

CREATE SYNONYM statement 4-27

CREATE TABLE SPACE statement 4-35, C-6

CREATE TABLE statement 4-29

CREATE VIEW statement 4-40, 4-41

C-style comments 1-13

cursor name, definition 2-5

cursors
- closing 4-19
- declaring 4-42
- opening 4-72

D

data field, definition 3-17

data type, definition 3-1

data types, general
- database 3-2
- host language 3-21
- loader 3-17
- local database 3-16
- unloader 3-17

data types, specific
- BINARY 3-3
- BLOB 3-4
- BLOB_FILE 3-22
- BLOB_LOCATOR 3-22
- CHARACTER 3-5
- CLOB 3-6
- CLOB_FILE 3-22
- CLOB_LOCATOR 3-22
- DATE 3-7
- DATE EXTERNAL 3-25
- DECIMAL 3-8
- DOUBLE PRECISION 3-7, 3-8
- FLOAT 3-7, 3-8
- int16_t 3-23
- int32_t 3-23
- int64_t 3-23
- int8_t 3-23
- INTEGER 3-8
- NUMERIC 3-8
- RAW 3-18
- REAL 3-9
- SMALLINT 3-9
- TIME 3-10
- TIMESTAMP 3-11
- TIMESTAMP EXTERNAL 3-25
- types for int and long 3-23
- uint16_t 3-23
- uint32_t 3-23
- uint64_t 3-23
- uint8_t 3-23
- VARBINARY 3-12
- VARBYTE 3-19
- VARCHAR 3-13
- VARRAW 3-19

database
- component names 2-6
- data types and functions 3-14
- data types, definition 3-2
- names 2-6

date and time input formats 3-19

DATE data type 3-7

date format string 2-15

date literal, definition 2-12

DAYOFMONTH scalar function 6-19

DAYOFWEEK scalar function 6-20

DAYOFYEAR scalar function 6-21

DAYS scalar function 6-22

DB2 database data types 3-17

DBA privilege 4-68

DECIMAL data type 3-8

decimal literal, definition 2-14

DECIMAL results in arithmetic operations 2-22

DECLARE statement 4-42

declaring
- arrays 4-14
- cursors 4-42
- host variables 4-12
- indicator variables 4-12
- new data types for variables and arrays 4-16

DECODE scalar function 6-23

default definition, user table 4-31

definitions
- account ID 2-4
- aggregate function 6-1
- alias name 2-5
- argument 4-1
- arithmetic operator 2-21
- auto-join 1-8
- basic predicate 5-3
- BETWEEN predicate 5-9
- boolean operator 5-5
- Cartesian product 1-8
- character literal 2-11
- comparison operator 2-23
- complex predicate 5-5
- component 4-1
- concatenation operator 2-23
- cursor name 2-5
- data field 3-17
- data type 3-1
- database data type 3-2
- database name 2-6
- date literal 2-12
- decimal literal 2-14
- dynamic SQL 1-4
- embedded SQL 1-3
- equi-join 1-8
- EXISTS predicate 5-10
- extraction 1-7
- file reference variable 2-8
- floating-point literal 2-13
- full table scan 1-7
- function 2-2, 6-1
- hexadecimal literals 2-14
- host language data type 3-21
- host variable 2-7
- hybrid IN 1-9
- IN predicate 5-11
- inner-join 1-7
- input data record 3-17
- integer literal 2-13
- join 4-94
- join table operation 1-7
- keyword 2-1
- left outer-join 1-7
- LIKE predicate 5-12
- literal 2-11
- loader data type 3-17
- locator variable 2-7
- logical operator 5-5
- nested loop 1-8
- nested query 1-9

NULL predicate 5-14
NULL value 2-24
ODBC 1-2
place holder 2-8
predicate 2-2
quantified predicate 5-7
restrictive clause 2-2
scalar function 6-2
select table operation 1-6
self-join 1-8
space 1-12
special register 2-25
SQL 1-1
SQL identifier 2-4
static SQL 1-3
StorHouse SQL 1-1
string of characters 3-5
subquery 1-9
substitution markers 2-8
time literal 2-12
timestamp literal 2-13

DELETE privilege 4-68

DELETE statement 4-44

DESCRIBE BIND VARIABLES FOR clause 4-46

DESCRIBE SELECT LIST clause 4-47

DESCRIBE statement 4-46

DISCONNECT statement 4-49

DISTINCT argument, SELECT statement 4-83

Distributed Relational Database Architecture (DRDA) 1-2

DOUBLE PRECISION data type 3-7, 3-8

DRDA (Distributed Relational Database Architecture) 1-2

DROP INDEX statement 4-50

DROP SYNONYM statement 4-52

DROP TABLE SPACE statement 4-54

DROP TABLE statement 4-53

DROP VIEW statement 4-56

dynamic SQL 1-4, 4-2

E

embedded SQL 1-3

Embedded SQL Interface (ESQL) 1-3

END DECLARE SECTION statement 4-11

equi-join, definition 1-8

error codes, list A-1

ESQL 1-3

EXECUTE IMMEDIATE statement 4-60

EXECUTE statement 4-57

EXISTS predicate 5-10

expression, definition 2-3

extractor 1-7

F

FETCH statement 4-62

file access group, StorHouse 4-9, 4-37, C-9

file reference variable 2-8

file set, user tablespace C-2

fixed-length data type 3-3

FLOAT data type 3-7, 3-8

FOR clause 4-93

format string
- definition 2-14
- for dates 2-15
- for times 2-16

FREE LOCATOR statement 4-66

FROM clause 4-86

full table scan 1-7

function, definition 2-2, 6-1

functions and database data types 3-14

functions, specific
- ABS 6-5
- ADD_MONTHS 6-7
- ASCII 6-8
- AVG 6-9
- BIT_LENGTH 6-11
- BLOB 6-12
- CHAR_LENGTH 6-13
- CHR 6-14
- CLOB 6-15
- CONCAT 6-16
- COUNT 6-18
- DAYOFMONTH 6-19
- DAYOFWEEK 6-20
- DAYOFYEAR 6-21
- DAYS 6-22
- DECODE 6-23
- GREATEST 6-25
- HOUR 6-26
- INITCAP 6-27
- INSTR 6-28
- LAST_DAY 6-30
- LEAST 6-31
- LENGTH 6-32
- LOWER 6-33
- LPAD 6-34
- LTRIM 6-36
- MAX 6-37
- MIN 6-38
- MINUTE 6-39
- MONTH 6-40
- MONTHS_BETWEEN 6-41
- NEXT_DAY 6-42
- NVL 6-43
- OCTET_LENGTH 6-44
- OVERLAY 6-45
- POSITION 6-47
- QUARTER 6-48
- RPAD 6-49
- RTRIM 6-51
- SECOND 6-52
- SUBSTR 6-53
- SUM 6-55
- TO_CHAR 6-57
- TO_DATE 6-58
- TO_HEX 6-59
- TO_NUMBER 6-61
- TO_TIME 6-62
- TRANSLATE 6-63
- TRIM 6-65
- UPPER 6-67
- WEEK 6-68
- YEAR 6-69

G

gateway, database 1-2

GRANT statement 4-67

GREATEST scalar function 6-25

GROUP BY clause 4-89

group, StorHouse file access C-9

H

hash index 4-25

HAVING clause 4-90

hexadecimal literal, definition 2-14

host language data types 3-21

host variable name 2-7

host variables, declaring 4-12

HOUR scalar function 6-26

hybrid IN 1-8

I

IBM Distributed Relational Database Architecture (DRDA) 1-2

identifier, definition 2-6

IN predicate 5-11

INDEX privilege 4-68

indexes
- assigning FSETS C-7
- assigning to a user tablespace 4-25
- assigning VSETs C-7
- creating 4-24
- dropping 4-50, 4-53

indicator variables, declaring 4-12

INITCAP scalar function 6-27

inner-join, definition 1-7

input data record, definition 3-17

input formats for date and time data 3-19

input host variable, describing 4-46

INSERT privilege 4-68

INSERT statement 4-70

INSTR scalar function 6-28

int16_t data type 3-23

int32_t data type 3-23

int64_t data type 3-23

int8_t data type 3-23

integer literal, definition 2-13

INTO clause
- DESCRIBE 4-46
- SELECT 4-85

invoking StorHouse SQL 1-2

ISQL 1-2

J

join columns 1-7

join condition 1-7, C-12

join operations 1-8

join, definition 1-7, 4-94

K

keyword
- definition 2-1

reserved B-1

L

LAST_DAY scalar function 6-30

LEAST scalar function 6-31

left outer-join, definition 1-7

LENGTH scalar function 6-32

LIKE predicate 5-12

list of
- aggregate functions 6-2
- loader data types 3-14
- reserved words B-1
- scalar functions 6-2
- SQL statements 4-2
- status codes A-1

literal, definition 2-11

literals
- character 2-11
- conversion of 3-28
- date 2-12
- floating-point 2-13
- hexadecimal 2-14
- integer 2-13
- time 2-12
- timestamp 2-13

loader data type
- conversion 3-23
- definition 3-17
- list of 3-18

LOB locator 2-7

local database 1-2

local database data types 3-16

locator variable, LOB 2-7

logical operator, definition 5-5

logical operators 2-20

LOWER scalar function 6-33

LPAD scalar function 6-34

LTRIM scalar function 6-36

M

masks
- date format strings 2-15
- time format strings 2-16

MAX aggregate function 6-37

MIN aggregate function 6-38

MINUTE scalar function 6-39

MONTH scalar function 6-40

MONTHS_BETWEEN scalar function 6-41

N

names, user-supplied 2-4

naming conventions
- account ID 2-4
- alias name 2-5
- cursor name 2-5
- database component name 2-6
- database name 2-6
- host variable name 2-7

nested loop 1-8

nested query, definition 1-9

NEXT_DAY scalar function 6-42

NOT logical operator 2-20

NOT NULL constraint 2-24

NULL predicate 5-14

NULL value, definition 2-24

NUMERIC data type 3-8

NVL scalar function 6-43

O

object identifier 3-4

OCTET_LENGTH scalar function 6-44

OID 3-4

Open Database Connectivity (ODBC) gateway 1-2

OPEN statement 4-72

operators
- arithmetic 2-21
- comparison 2-23
- set 2-18

OR boolean operator 5-5

OR logical operator 2-20

ORDER BY clause 4-91

output host variables, describing 4-46

OVERLAY scalar function 6-45

P

padding
- bit data 3-3
- character data 3-5

place holder, definition 2-8

POSITION scalar function 6-47

precompiler, ESQL 1-4

predicate order 5-2

predicate, definition 2-2

predicates, specific
- basic 5-3
- BETWEEN 5-9
- complex 2-20, 5-5
- EXISTS 5-10
- IN 5-11
- LIKE 5-12
- NULL 5-14
- quantified 5-7

PREPARE statement 4-75

privileges 4-68

PUBLIC account 4-69

Q

quantified predicate 5-7

QUARTER scalar function 6-48

queries, types of 1-6

R

range index 4-25

RAW data type 3-18

REAL data type 3-9

rescaling DECIMAL columns 3-24

RESOURCE privilege 4-68

restrictive clause, definition 2-2

result data types
- concatenation 2-24
- UNION set operator 2-18

result table, definition 1-6

REVOKE statement 4-77

right outer-join, definition C-13

ROLLBACK WORK statement 4-80

RPAD scalar function 6-49

RTRIM scalar function 6-51

S

scalar function, definition 6-2

scalar functions, specific
- ABS 6-5
- ADD_MONTHS 6-7
- ASCII 6-8
- BIT_LENTH 6-11
- BLOB 6-12
- CHAR_LENGTH 6-13
- CHR 6-14
- CLOB 6-15
- CONCAT 6-16
- DAYOFMONTH 6-19
- DAYOFWEEK 6-20
- DAYOFYEAR 6-21
- DAYS 6-22
- DECODE 6-23
- GREATEST 6-25
- HOUR 6-26
- INITCAP 6-27
- INSTR 6-28
- LAST_DAY 6-30
- LEAST 6-31
- LENGTH 6-32
- LOWER 6-33
- LPAD 6-34
- LTRIM 6-36
- MINUTE 6-39
- MONTH 6-40
- MONTHS_BETWEEN 6-41
- NEXT_DAY 6-42
- NVL 6-43
- OCTET_LENGTH 6-44
- OVERLAY 6-45
- POSITION 6-47
- QUARTER 6-48
- RPAD 6-49
- RTRIM 6-51
- SECOND 6-52
- SUBSTR 6-53
- TO_CHAR 6-57
- TO_DATE 6-58
- TO_HEX 6-59
- TO_NUMBER 6-61
- TO_TIME 6-62
- TRANSLATE 6-63
- TRIM 6-65
- UPPER 6-67
- WEEK 6-68
- YEAR 6-69

SCAN privilege 4-68

SECOND scalar function 6-52

SELECT privilege 4-68

SELECT statement
- description 4-82
- FOR clause 4-93
- FROM clause 4-86
- GROUP BY clause 4-89
- HAVING clause 4-90
- INTO clause 4-85
- ORDER BY clause 4-91
- WHERE clause 4-88

select table operation, definition 1-6

self-join, definition 1-8

SET clause 4-106

SET CONNECTION statement 4-105

set operator 2-18

SMALLINT data type 3-9

SOME keyword, quantified predicate 5-7

space, definition 1-12

special register, definition 2-25

special registers
- SYSDATE 2-25
- SYSTIME 2-26
- USER 2-25

SPUFI 1-2

SQL identifier 2-4

SQL status codes A-1

SQL, invoking 1-2

SQL_IDX_TYPE system parameter 4-24

SQLCA 4-60

SQLDA 4-46

SQL-style comments 1-13

statements, list of 4-2

statements, specific
- ALTER TABLE SPACE 4-7, C-2
- BEGIN DECLARE SECTION 4-11
- CLOSE 4-19
- COMMIT WORK 4-20
- CONNECT 4-22
- CREATE INDEX 4-24
- CREATE SYNONYM 4-27
- CREATE TABLE 4-29
- CREATE TABLE SPACE 4-7, 4-35, C-6
- CREATE VIEW 4-40, 4-41
- DECLARE 4-42
- DELETE 4-44
- DESCRIBE 4-46
- DISCONNECT 4-49
- DROP INDEX 4-50
- DROP SYNONYM 4-52
- DROP TABLE 4-53
- DROP TABLE SPACE 4-54
- DROP VIEW 4-56
- END DECLARE SECTION 4-11
- EXECUTE 4-57
- EXECUTE IMMEDIATE 4-60
- FETCH 4-62
- FREE LOCATOR 4-66
- GRANT 4-67
- INSERT 4-70
- OPEN 4-72
- PREPARE 4-75
- REVOKE 4-77
- ROLLBACK WORK 4-80
- SELECT 4-82
- SELECT (RIGHT OUTER-JOIN) C-13

SET CONNECTION 4-105
UPDATE 4-106
VALUES INTO 4-108
WHENEVER 4-109

static SQL 1-3

status codes, list A-1

STH access group 4-37, C-9

StorHouse xix

StorHouse account ID 4-22

StorHouse password 4-22

StorHouse SQL reserved words B-1

StorHouse SQL, definition 1-1

StorHouse/Admin 1-2

StorHouse/RM xx

StorHouse/SM xix

string of characters, definition 3-5

subquery, definition 1-9

substitution marker, definition 2-8

SUBSTR scalar function 6-53

SUM aggregate function 6-55

synonyms
creating 4-27
dropping 4-52

syntax rules for StorHouse SQL 1-10

SYSDATE special register 2-25

SYSSMUSERS system table 2-3

system parameter, SQL_IDX_TYPE 4-24

system table, SYSSMUSERS 2-3

SYSTIME special register 2-26

T

tables
creating 4-29
dropping 4-53

TIME data type 3-10

time format string 2-16

time literal, definition 2-12

TIMESTAMP data type 3-11

timestamp literal, definition 2-13

TO_CHAR scalar function 6-57

TO_DATE scalar function 6-58

TO_HEX scalar function 6-59

TO_NUMBER scalar function 6-61

TO_TIME scalar function 6-62

TRANSLATE scalar function 6-63

TRIM scalar function 6-65

type declarations 4-12

U

uint16_t data type 3-23

uint32_t data type 3-23

uint64_t data type 3-23

uint8_t data type 3-23

unary arithmetic operator 2-22

UNION ALL set operator 2-18

UNION set operator 2-18

unloader data type
    conversion 3-26
    definition 3-17

UPDATE privilege 4-68

UPDATE statement 4-106

UPPER scalar function 6-67

USER special register 2-25

user tables
    creating 4-29
    dropping 4-53

user tablespaces
    creating 4-35
    dropping 4-54

user-supplied names 2-4

USING clause
    EXECUTE 4-57, 4-58
    OPEN 4-72

USING DESCRIPTOR clause
    EXECUTE 4-57, 4-58
    FETCH 4-63
    OPEN 4-72

V

value index 4-25

VALUES INTO statement 4-108

VARBINARY data type 3-12

VARBYTE data type 3-19

VARCHAR data type 3-13

variable declarations 4-11

variable-length data type 3-2

VARRAW data type 3-19

views
    creating 4-40
    dropping 4-56

W

WEEK scalar function 6-68

WHENEVER statement 4-109

WHERE clause 4-88

WITH GRANT OPTION 4-69

Y

YEAR scalar function 6-69