



StorHouse ESQL Manual

StorHouse/RM Release 3.2

Publication Number
900121 Rev. H

March 21, 2002





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: 900121 Rev. H

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 Unpublished-rights reserved under the copyright laws of the United States.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.

Contents

Welcome	ix
StorHouse family of products	ix
StorHouse/SM	ix
StorHouse/RM	x
Control Center	x
Purpose of this document	x
Intended audience	xi
Contents	xi
Related documentation	xii
Format conventions	xiv
 Chapter 1: Introducing StorHouse ESQL	 1-1
Understanding the ESQL precompiler	1-1
Learning basic concepts	1-2
Embedded, or static, SQL	1-3
Types of embedded SQL statements	1-3
Static versus dynamic SQL	1-4
NULL values	1-5
Host language variables	1-5
Locator variables	1-6
File reference variables	1-6

Indicator variables	1-6
Data types	1-7
Cursors	1-7
Errors and warnings	1-7
Guidelines for embedding SQL in C	1-8
Comments	1-8
SQL-style comment	1-8
C-style comment	1-9
C++-style comments	1-9
Continuations	1-10
Delimiters	1-10
Host variable names	1-10
Operators	1-11
Statement labels	1-11

Chapter 2: Satisfying program requirements2-1

Coding a Declare Section	2-1
Declaring variables and types	2-2
Declaring variables and types as StorHouse or C data types	2-4
Mapping StorHouse data types to C language types	2-4
Defining StorHouse data types	2-6
Declaring variables with type definitions	2-20
Declaring variables as host arrays	2-21
ESQL format	2-22
C format	2-23
Declaring an array as a new data type	2-24
Using host variables	2-25
Using indicator variables	2-27
Managing connectivity to StorHouse	2-28
Connecting to a StorHouse database	2-29
Changing the current connection	2-30
Terminating a connection	2-31

Chapter 3: Submitting queries in ESQL	3-1
About queries	3-1
Queries that return a single row	3-2
Queries that return multiple rows	3-3
Using cursors	3-4
Associating a cursor with a query	3-5
Opening a cursor	3-6
Retrieving rows using a cursor	3-7
Closing a cursor	3-9
Using a cursor with host variable arrays	3-9
 Chapter 4: Handling errors and warnings	 4-1
Using the SQLCA	4-1
SQLCA structure definition	4-2
Checking the sqlcode for status codes	4-4
Checking for warnings	4-5
Using WHENEVER	4-6
Scope of WHENEVER	4-7
 Chapter 5: Using dynamic SQL	 5-1
About dynamic SQL	5-1
Storing dynamic SQL as a character string	5-2
Understanding substitution markers	5-3
Correlating substitution markers with host variables	5-3
Example	5-3
Scenarios for using dynamic SQL	5-4
Scenario 1: Non-SELECT without markers	5-5
Scenario 2: Non-SELECT with markers	5-6
About PREPARE	5-7

About EXECUTE	5-8
Associating markers with host variables	5-8
Example	5-9
Scenario 3: Fixed-list SELECTs	5-10
Scenario 4: SELECT using an SQLDA	5-12
About an SQL descriptor area (SQLDA)	5-12
Storing information in an SQLDA	5-13
Understanding the SQLDA structure definition	5-14
Setting values of sqlvln32 and sqlvtype fields	5-18
Resetting, or coercing, data types	5-19
Checking space for SQLDA entries	5-19
Checking space for variable name data	5-20
Allocating an SQLDA	5-21
Allocating variable entries in an SQLDA	5-22
Initializing storage as an SQLDA	5-22
Setting values in an SQLDA variable entry	5-23
Freeing an SQLDA	5-24
Checking the size of your SQLDA	5-24
Allocating SQLDA buffers for data and indicator variables	5-27
Calculating the buffer size (tpe_da_getbsize)	5-27
Initializing buffer pointers (tpe_da_setptrs)	5-28
Using multiple SQLDAs	5-29
Reusing the same SQLDA	5-29
Understanding the status value	5-29
Reviewing the basics	5-30
Satisfying individual program requirements	5-31
Using an SQLDA for array fetches	5-32
About the standard method	5-32
About the pointer-fetch method	5-33
Sample program	5-34
Chapter 6: Accessing large objects	6-1
Ways to access LOB values	6-1
Locator variables	6-2

File reference variables	6-2
Sample LOB value	6-3
Placing LOB data into a host variable	6-3
Using a locator variable to select LOB data	6-4
Declaring a locator variable	6-5
Issuing the query	6-6
Example using SELECT INTO	6-6
Example using FETCH	6-6
Manipulating a LOB value through a locator variable	6-7
Releasing a locator variable	6-8
Placing LOB data into a client file	6-8
Declaring a file reference variable	6-9
Initializing the client file variable	6-9
Issuing the query	6-10
Example using SELECT INTO	6-10
Example using FETCH	6-12

Chapter 7: Using the StorHouse extractor 7-1

About the StorHouse extractor	7-1
Types of eligible queries	7-2
Simple queries	7-2
Format	7-2
Example	7-2
Full segment queries	7-3
Format	7-3
Example	7-3
Qualifying for extractor processing	7-4
Checking the SQLCA	7-5

Chapter 8: Managing transactions	8-1
About transactions	8-1
Starting a transaction	8-2
Ending a transaction	8-3
Committing a transaction	8-3
Committing DDL statements	8-4
Committing non-DDL statements	8-4
Rolling back a transaction	8-4
Automatic rollback	8-5
Locking	8-5
Types of locks	8-5
Shared locks	8-6
Exclusive locks	8-6
Duration of a lock	8-7
 Chapter 9: Using the ESQL precompiler	 9-1
Setting environment variables	9-1
Issuing the esqlc command	9-3
 Appendix A: Reviewing a sample program	 A-1
 Appendix B: Converting and comparing data	 B-1
 Appendix C: Developing ESQL applications	 C-1
 Index	

Welcome

The *StorHouse Embedded Structured Query Language (ESQL)* enables development of C and C++ applications that contain embedded Structured Query Language (SQL) statements. These applications can use StorHouse SQL to access database data in StorHouse databases.

StorHouse family of products

StorHouse[®] is the FileTek[®] enterprise-wide solution for managing the capture, storage, movement, and access of gigabytes (GB) 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/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 in the StorHouse storage hierarchy, including tape. 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, 2000, and XP Pro 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 ESQL Manual* describes how to use SQL statements in a C or C++ application and how to prepare and execute those statements dynamically. Other topics include error handling methods, the StorHouse precompiler, and the StorHouse extractor feature. This manual also provides ESQL examples and a sample program.

Intended audience

The *StorHouse ESQL Manual* is intended for the people who write StorHouse database applications at your site. These applications may perform functions for end-user departments or implement basic StorHouse database administration tasks.

This manual is designed for experienced programmers. It assumes that the audience is already proficient in C and/or C++, database application design, database administration, StorHouse fundamentals, and SQL.

Contents

This document is organized as follows:

- Chapter 1, “Introducing StorHouse ESQL,” discusses basic concepts that you need to understand before you begin developing ESQL applications. It also provides guidelines for embedding SQL in C programs.
- Chapter 2, “Satisfying program requirements,” explains the programming techniques that enable communication between your ESQL application and StorHouse.
- Chapter 3, “Submitting queries in ESQL,” explains how to submit queries using embedded SQL. It explains the SELECT INTO statement, cursors, and array fetches.
- Chapter 4, “Handling errors and warnings,” explains how to use the SQL communications area (SQLCA) and the WHENEVER statement to manage error and warning conditions.
- Chapter 5, “Using dynamic SQL,” defines dynamic SQL and provides four scenarios for using dynamic SQL in applications. It also explains how to allocate and use an SQL descriptor area (SQLDA).

- Chapter 6, “Accessing large objects,” explains how to access large objects (LOBs) using locator variables and file reference variables.
- Chapter 7, “Using the StorHouse extractor,” defines the extractor feature and explains the types of queries that are eligible for extractor processing.
- Chapter 8, “Managing transactions,” explains how to start transactions and end them with COMMIT WORK or ROLLBACK WORK.
- Chapter 9, “Using the ESQL precompiler,” discusses the requirements for precompiling, compiling, and linking ESQL programs and how to use the esqlc command with the proper command options.
- Appendix A, “Reviewing a sample program,” contains a sample program that illustrates how to code static queries.
- Appendix B, “Converting and comparing data types,” explains the StorHouse API calls tpe_conv_data and tpe_compare_data.
- Appendix C, “Developing ESQL applications,” contains useful tips and guidelines for writing ESQL applications.

Related documentation

You should be familiar with these manuals in the StorHouse/RM User Document Set:

- The *StorHouse SQL Reference Manual*, publication number 900111, is the comprehensive reference for StorHouse SQL. It defines StorHouse SQL statements, functions, predicates, and data types. In addition, it contains format descriptions, examples, and a list of all StorHouse status codes.

- The *StorHouse SQL Quick Reference*, publication number 900122, is a stand-alone mini-manual that contains formats and examples of all StorHouse SQL statements, functions, and predicates.
- The *StorHouse Database Administration Guide*, publication number 900108, defines StorHouse database concepts and explains how to perform StorHouse database administration tasks like creating user tables and indexes, managing accounts and privileges, and defining user tablespaces.
- The *StorHouse/RM Glossary*, publication number 900112, defines terms used in StorHouse/RM User Document Set.

In addition, if you are performing StorHouse system administration tasks, you may find the following manuals helpful:

- The *StorHouse Glossary*, publication number 900027, defines the terminology used in FileTek StorHouse publications. This manual is intended for all users.
- The *StorHouse Concepts and Facilities Manual*, publication number 900026, defines the basic concepts, structures, and functions of StorHouse. This manual is intended for all users.
- The *Command Language Reference Manual*, publication number 900005, contains descriptions of StorHouse Command Language commands and is intended for all users.
- The *System Administrator's Guide*, publication number 900007, describes system recovery, account administration, and storage management procedures and concepts for StorHouse. This guide is intended for the StorHouse system administrator.
- The *Messages and Codes Manual*, publication number 900032, lists all StorHouse system and host software messages. This manual is intended for all users.

Format conventions

This manual uses the Courier font (this is Courier font) to represent program code and the following format conventions to illustrate StorHouse SQL syntax.

Convention	Description
UPPERCASE	Uppercase terms indicate keywords that are part of the syntax. Type keywords in any case.
lowercase	Lowercase terms refer to grammar elements and user-supplied values. When supplying values, only quoted strings are case sensitive.
(), / * - ; : . + '	These characters are part of the syntax. Type them as shown.
{ }	Braces indicate that the item is required. When a list of items is enclosed in braces and separated by a vertical bar, you must choose one item.
[]	Brackets indicate that the item is optional.
	Vertical bar separates alternatives. You can specify one of the alternatives shown.
...	Ellipsis points indicate that you can repeat the part of the statement preceding them any number of times.

The following statement (a simplified query) uses the format 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 syntax. The braces indicate a requirement, while the vertical bar indicates a choice of items within the braces.
- owner and the WHERE clause are enclosed in square brackets ([]) because they are not required to complete this statement. They are optional.



Welcome

Format conventions

Introducing StorHouse ESQL

The StorHouse Structured Query Language (SQL) provides an *Embedded SQL Interface (ESQL)* that supports coding SQL statements in C and C++ programs. By embedding SQL statements in a host program, you can develop applications that are more flexible than those developed in just the host language or SQL.

This chapter discusses:

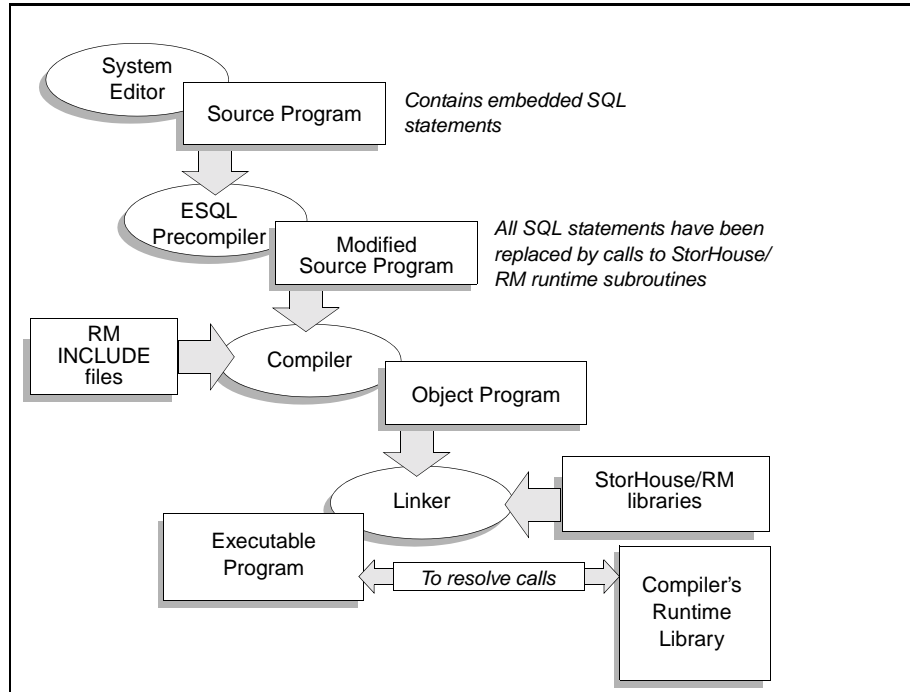
- The ESQL precompiler
- Basic ESQL concepts
- Guidelines for embedding SQL in C programs

Understanding the ESQL precompiler

The *StorHouse ESQL precompiler* lets you embed SQL statements in a C or C++ source program. StorHouse/RM uses the StorHouse ESQL precompiler to build database applications in the SQL development environment.

The ESQL precompiler accepts your source program as input and then translates the embedded SQL into host language statements that include StorHouse/RM runtime subroutines. The output of this translation is a pure C or C++ program, which you can compile, link, and execute. The ESQL precompiler also accepts C or C++ object files and passes them to the C or C++ linker.

The following diagram illustrates the path from source code to executable for an ESQL program.



Learning basic concepts

You should understand these basic concepts before you begin developing ESQL applications:

- Static (embedded) and dynamic SQL
- Types of embedded SQL statements
- Use of NULL values
- Use of host language and indicator variables in ESQL programs
- Database and host language data types

- The available tools for handling errors and warnings
- Cursors

See other chapters in this manual for more detailed information about these topics.

Embedded, or static, SQL

Embedded, or static, SQL statements are SQL statements that are coded within a C or C++ application. The application that contains the SQL is the *host program*. The language of the host program is the *host language*. You can mix SQL statements and host language statements in an ESQL program. In addition, you can use host language variables in embedded SQL statements.

It's easy to recognize static SQL statements because they are always preceded by the keywords EXEC SQL and followed by a semicolon, which is the statement terminator for C. SQL statements bracketed by the keywords EXEC SQL and a semicolon are called *ESQL constructs*.

The following sample code illustrates an SQL construct:

```
EXEC SQL
    SELECT emp_no,dept_id
    INTO :emp_num_v, :dept_id_v
    FROM emptable ;
```

Types of embedded SQL statements

There are two types of embedded SQL statements: declarative and executable. ESQL *declarative statements* bracket DECLARE SECTIONS, define host variables and indicator variables, and provide branching instructions for error processing. They do not generate any StorHouse/RM calls.

The StorHouse declarative SQL statements are:

- BEGIN DECLARE SECTION
- END DECLARE SECTION
- WHENEVER

Executable statements generate StorHouse/RM calls and return codes. They execute instructions on a specified database at runtime. Executable statements access the database and form the body of an ESQL program. They can change the state of a database.

You can group executable statements into the following categories:

Statement type	Used to
Data Manipulation (DML)	Perform data manipulation operations like SELECT, INSERT, UPDATE and DELETE. In StorHouse, INSERT, UPDATE, and DELETE may only be used to manipulate system tables and system table views.
Data Definition (DDL)	Create and remove data definitions in a given database (for example, create user tables, indexes for user tables, and system table views).
Transaction Management	Manage SQL transactions (for example, COMMIT WORK and ROLLBACK WORK).

Static versus dynamic SQL

You use static SQL when you know your SQL statements at 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 things that may change from one execution to the next are the host variables in your search conditions.

Dynamic SQL is more complicated and harder to program than static SQL. You use dynamic SQL when you don't know which SQL statements you intend to execute or which columns and tables you plan to manipulate until runtime. You typically use a special structure called an *SQL descriptor area (SQLDA)* with dynamic SQL. The SQLDA provides your program with information about the variables in

your SQL statements. See Chapter 5, “Using dynamic SQL,” for more information about an SQLDA.

NULL values

NULL means different things in SQL and C. In StorHouse, a *NULL* value is a value that is unknown, not applicable, or missing. It's not the same as a numeric zero, a string of blanks, or a variable-length string of length zero.

A database column of any data type can have a NULL value. In fact, NULL is the default value for a column, provided that column definition in CREATE TABLE does not contain the DEFAULT definition clause or the NOT NULL column constraint. Refer to the *StorHouse SQL Reference Manual* for more information about CREATE TABLE.

StorHouse/RM uses the NVL function, the IS [NOT] NULL comparison operator, and indicator variables to manage NULL values.

Host language variables

Host language variables enable communication between StorHouse SQL and your application. A *host language variable* is an application variable that host language statements and embedded SQL statement can reference. You declare host variables in a required program component called a *Declare Section*.

- *Input host variables* pass data to StorHouse/RM. They are typically used in WHERE and HAVING clauses.
- *Output host variables* pass data and status information to your program. They are typically used in the INTO clauses of the SELECT and FETCH statements as well as in the VALUES INTO statement.

You can refer to and manipulate LOB values using host variables just as you would any other type of data. Host variables, however, use the client memory

buffer which may not be large enough to hold LOB values. StorHouse/RM provides two types of host variables for accessing and manipulating LOB values:

- Locator variables
- File reference variables

Locator variables

A *locator variable* is a type of host variable you use to identify and manipulate a LOB value or part of a LOB value at the StorHouse server. When you use a locator variable, the LOB value remains on the server and the locator moves to the client. The value associated with the locator is valid until you end the transaction (COMMIT WORK or ROLLBACK WORK statement) or explicitly release the locator (FREE LOCATOR statement). You define locator variables with the BLOB_LOCATOR and CLOB_LOCATOR data types.

File reference variables

A *file reference variable* is a type of host variable you use to place a LOB value or part of it in a client file without going through the application's memory. The file reference variable contains the name of the client file. The file referenced by the file reference variable must be accessible to the system on which the program runs. Currently, you can use file reference variables as output only, that is, you can transfer a LOB value from StorHouse/RM to a client file. You define file reference variables with the BLOB_FILE and CLOB_FILE data types.

Indicator variables

You can associate a host variable with an optional indicator variable. An *indicator variable* is a short variable that detects NULL or truncated values. You declare indicator variables in a required program component called a *Declare Section*.

Data types

StorHouse recognizes database and host language data types. *Database data types* define how StorHouse/RM stores information in database columns. *Host language data types* define the data storage format for host variables. C language structures represent the StorHouse BLOB, BLOB_FILE, CLOB, CLOB_FILE, DATE, DECIMAL (synonym for NUMERIC), TIME, and TIMESTAMP data types. See “Defining StorHouse data types” on page 2-6 for descriptions of the data types you can use in your applications.

Cursors

ESQL uses a cursor to process the rows that satisfy your queries. These rows are called a *result set*, or *active set*. A *cursor* is a named structure that points to a specific row within a result set. The size of the result set depends on the number of rows that satisfy the query search condition. The row currently being processed by the cursor is called the *current row*.

You must process queries that retrieve more than one row by:

- Explicitly *declaring*, or naming, a cursor for the SELECT statement
- Opening the cursor to execute the query and identify the result set
- Using the cursor to *fetch*, or retrieve, each row in the result set
- Closing the cursor when there are no more rows to be fetched

Errors and warnings

StorHouse/RM provides two ways to handle errors and warnings: the SQL communications area (SQLCA) and the WHENEVER statement. An *SQLCA* is a structure that provides an application with status information about the most recently executed SQL statement. C programs implement the SQLCA as a global structure that the ESQL precompiler automatically declares and defines.

You can check the SQLCA for information such as:

- Warning flags
- Error (status) codes
- Diagnostic text
- Number of rows processed

The SQLCA field, `sqlcode`, indicates status information as follows:

- 0 – successful execution
- Positive – no more rows to be fetched (value of 100 for `SQL_NOT_FOUND`)
- Negative – an error occurred

The `WHENEVER` statement provides added flexibility because it lets you supply specific instructions for error processing. With `WHENEVER`, you can stop program execution, continue with the next program statement, or branch to a specified host language label depending on specific exceptions.

Guidelines for embedding SQL in C

This section provides guidelines for embedding StorHouse SQL statements in C programs. These guidelines include syntax rules, restrictions, and coding conventions. They are listed in alphabetical order for your convenience.

Comments

You can include SQL-style, C-style, and C++-style comments in StorHouse SQL statements.

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 ;
```

Continuations

C dictates the rules for continuing SQL statements from one line to the next. To continue a string literal in an SQL statement from one line to another, use a backslash (\) as the last character on the line.

Delimiters

The keywords EXEC SQL and the semicolon (;) delimit embedded SQL statements. You must code the keywords EXEC SQL on the same line. The embedded SQL statement following EXEC SQL may begin on the same line or on subsequent lines.

StorHouse supports the use of apostrophes (') to enclose string literals and quotations (") to delimit identifiers. In the following example, apostrophes enclose the string literal 123, and quotations enclose the column identifier name.

```
EXEC SQL
    SELECT "name"
    FROM emp_table
    WHERE emp_id = '123';
```

Host variable names

You can use any valid C name as a host variable name. Refer to the documentation for your C compiler for your system's exact requirements.

Operators

The following table shows the differences between SQL and C operators. You cannot use these C operators in SQL statements:

Use these operators in SQL	Use these operators in C
NOT	!
AND	&&
OR	
=	==

Statement labels

You can include standard C statement labels in StorHouse SQL statements. The format of a label name is:

label_name:

The following example uses the statement label `finish_it`:

```
EXEC SQL
    WHENEVER NOT FOUND GO TO finish_it;
...
finish_it:
    EXEC SQL
    ...
```

Refer to the documentation for your C compiler for your system's exact requirements.

1

Introducing StorHouse ESQL

Guidelines for embedding SQL in C

Satisfying program requirements

This chapter explains the programming techniques that enable communication between your ESQL application and StorHouse/RM. Topics include:

- Coding a Declare Section
- Declaring variables
- Choosing data types for variable declarations
- Using host and indicator variables
- Handling NULL values
- Managing database connections from your program

Coding a Declare Section

A *Declare Section* is a required ESQL program component that contains your host language, indicator variable, and new type declarations. The ESQL precompiler generates the corresponding host language declarations for these variables and types so that you can use them at your convenience in SQL and C. ESQL does not recognize variables or types that are defined in C language statements coded outside a Declare Section.

StorHouse/RM uses the following ESQL constructs to mark the beginning and end of a Declare Section:

```
EXEC SQL BEGIN DECLARE SECTION ;  
/* your variable declarations go between these statements */  
EXEC SQL END DECLARE SECTION ;
```

Follow these rules when you code a Declare Section:

- A Declare Section must be located before any other ESQL constructs. Only C statements can precede a Declare Section.
- A Declare Section may contain declarations for host variables and indicator variables.
- You may declare local and global variables. The location of a Declare Section in your program determines a variable's scope.
- Your program may contain more than one Declare Section.
- A Declare Section cannot reference names declared in typedef statements.

The following Declare Section declares five host variables as host language char types and one indicator variable as host language short type:

```
EXEC SQL BEGIN DECLARE SECTION ;
    char emp_num_v ;
    char first_name_v [24] ;
    char initial_v [1] ;
    char last_name_v [40] ;
    char dept_id_v [8] ;
    short i_initial_v ;
EXEC SQL END DECLARE SECTION ;
```

Declaring variables and types

There are several ways to declare host and indicator variables in a Declare Section. You can declare:

- Variables as certain host language or StorHouse data types. StorHouse data types for source or target columns and host data types for variables must be

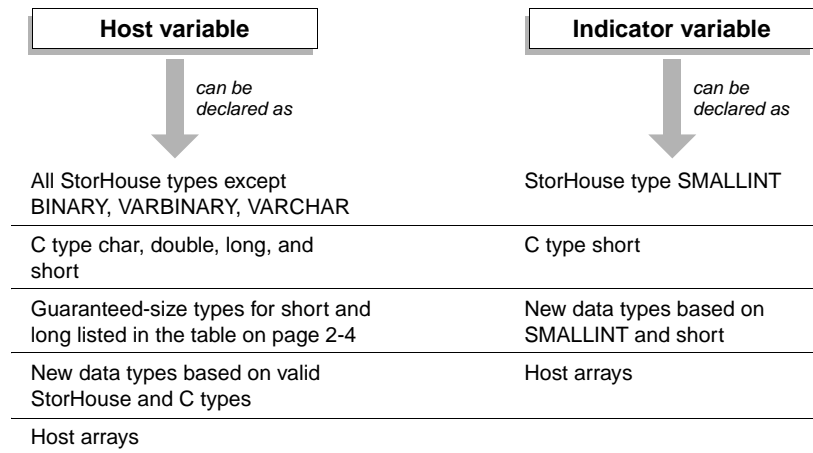
compatible but they do not have to match (for example, SMALLINT and long). See the table on page 2-4 for a list of compatible data types.

- New data types with the characteristics of host language or StorHouse data types; then declare host variables using your newly defined data type.
- Variables as host arrays

You *cannot* declare:

- Variables to be a structure type
- Pointers as host variables
- Variables as the StorHouse VARCHAR, BINARY, or VARBINARY data types

The following drawing summarizes the rules for variable declarations.



Declaring variables and types as StorHouse or C data types

You can declare host variables as the C data types `char`, `double`, `long`, and `short` and all StorHouse types except for `BINARY`, `VARBINARY`, and `VARCHAR`. StorHouse/RM rather than your application manages data conversions from one representation to the other. You can declare indicator variables as the C type `short` or the StorHouse type `SMALLINT`.

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. 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.

Other types for int and long	Description
<code>int64_t</code>	64-bit signed integer
<code>uint64_t</code>	64-bit unsigned integer
<code>int32_t</code>	32-bit signed integer
<code>uint32_t</code>	32-bit unsigned integer
<code>int16_t</code>	16-bit signed integer
<code>uint16_t</code>	16-bit unsigned integer
<code>int8_t</code>	8-bit signed integer
<code>uint8_t</code>	8-bit unsigned integer

Mapping StorHouse data types to C language types

The following table maps StorHouse data types to C language types and structures. Notice that the StorHouse data types `BLOB`, `BLOB_FILE`, `CLOB`, `CLOB_FILE`, `DATE`, `DECIMAL`, `TIME`, and `TIMESTAMP` are represented inter-

nally by C language structures. Descriptions of these data types include the C structure definition.

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

¹See the data type definitions of BINARY, VARBINARY, and VARCHAR for a list of restrictions that apply to these data types.

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

Defining StorHouse data types

The following tables define StorHouse data types.

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

BINARY specification

C language representation	unsigned char
Restrictions	<ul style="list-style-type: none"> ■ <i>Never</i> declare host variables as the BINARY data type. Instead, use an SQLDA and the DESCRIBE statement. See Chapter 5 for more information about SQLDAs and DESCRIBE. ■ The data in a BINARY field may contain 0 (NUL) bytes. Therefore never handle binary data as a null-terminated string.

BLOB The BLOB data type defines bit data as a variable-length array of bytes up to 2 GB in size.

BLOB specification

C language representation	tpe_blob_t or TPE_BLOB_DECL(x,y)
Declare BLOB variables as	BLOB(L) where (L) is the maximum length of the BLOB in bytes, K, M, or G
Example	<pre>EXEC SQL BEGIN DECLARE SECTION; BLOB(40K) part_picture; EXEC SQL END DECLARE SECTION;</pre>

The structure definition for a BLOB host variable is (where <hvn> is the host variable name and <sz> is the variable size):

```
struct {
    int32_t    <hvn>_reserved;
    uint32_t   <hvn>_length;
    char       <hvn>_data[<sz>];
} <hvn>;
```

Field	Description
<hvn>_reserved	Reserved for future use (must be 0).
<hvn>_length	The length of the BLOB value.
<hvn>_data	The BLOB data.

BLOB_FILE

The BLOB_FILE data type defines variable-length bit data stored in an external file. This data type defines a file reference variable for a BLOB value. The file reference variable expands into a type definition with four parts.

Note: You cannot use the BLOB_FILE and CLOB_FILE data types with a multi-row array-fetch operation. If these types are present, then the sqldnrow field of the SQLDA must be 1; otherwise, an error occurs. Also, you cannot use these data types with a pointer-fetch operation, even if sqldnrow is 1.

BLOB_FILE specification	
C language representation	tpe_blob_file_t
Declare BLOB file reference variables as	BLOB_FILE
Example	EXEC SQL BEGIN DECLARE SECTION; BLOB_FILE part_picture; EXEC SQL END DECLARE SECTION;

The structure definition for tpe_blob_file_t is:

```
typedef struct {
    uint32_t    name_length;
    uint32_t    data_length;
    uint32_t    file_options;
    char        name[255];
} tpe_blob_file_t;
```

Field	Description
name_length	The length of the file name that is stored in name.
data_length	The length of the data value in the file.
file_options	Options that control the use of the file. Valid values are: <ul style="list-style-type: none">■ TPE_FILE_CREATE – Creates a new file. An error occurs if the file exists.■ TPE_FILE_OVERWRITE – Replaces any existing file.■ TPE_FILE_APPEND – Appends fetched data to an existing file or creates a new file.
name	The name of the file.

BLOB_ LOCATOR

The BLOB_LOCATOR data type defines a locator variable for a BLOB value. An application may use this locator variable to manipulate BLOB values at StorHouse.

BLOB_LOCATOR specification	
C language representation	int32_t
Declare BLOB locator variables as	BLOB_LOCATOR
Example	EXEC SQL BEGIN DECLARE SECTION; BLOB_LOCATOR part_picture; EXEC SQL END DECLARE SECTION;

CHARACTER The CHARACTER (CHAR) data type corresponds to a fixed-length array of characters.

CHARACTER specification

C language representation	char
---------------------------	------

Declare CHARACTER variables as	char or CHAR
--------------------------------	--------------

Example	EXEC SQL BEGIN DECLARE SECTION ; char city_v [19] ; CHAR name_v [19] ; EXEC SQL END DECLARE SECTION ;
---------	--

CLOB The CLOB data type defines character data as a variable-length array of bytes up to 2 GB in size.

CLOB specification

C language representation	tpe_clob_t or TPE_CLOB_DECL(x,y)
---------------------------	----------------------------------

Declare CLOB variables as	CLOB(L) where (L) is the maximum length of the CLOB in bytes, K, M, or G
---------------------------	--

Example	EXEC SQL BEGIN DECLARE SECTION; CLOB(40K) part_description; EXEC SQL END DECLARE SECTION;
---------	---

The structure definition for a CLOB host variable is (where <hvn> is the host variable name and <sz> is the variable size):

```
struct {  
    int32_t    <hvn>_reserved;  
    uint32_t   <hvn>_length;  
    char       <hvn>_data[<sz>];  
} <hvn>;
```

Field	Description
<hvn>_reserved	Reserved for future use (must be 0).
<hvn>_length	The length of the CLOB value.
<hvn>_data	The CLOB data.

CLOB_FILE The CLOB_FILE data type defines variable-length character data stored in an external file. The file reference variable expands into a type definition with four parts.

Note: You cannot use the BLOB_FILE and CLOB_FILE data types with a multi-row array-fetch operation. If these types are present, then the sqldnrow field of the SQLDA must be 1; otherwise, an error occurs. Also, you cannot use these data types with a pointer-fetch operation, even if sqldnrow is 1.

CLOB_FILE specification	
C language representation	tpe_clob_file_t
Declare CLOB file reference variables as	CLOB_FILE
Example	EXEC SQL BEGIN DECLARE SECTION; CLOB_FILE part_description; EXEC SQL END DECLARE SECTION;

The structure definition for tpe_clob_file_t is:

```
typedef struct {
    uint32_t    name_length;
    uint32_t    data_length;
    uint32_t    file_options;
    char        name[255];
} tpe_clob_file_t;
```

Field	Description
name_length	The length of the file name that is stored in name.
data_length	The length of the data value in the file.

Field	Description
file_options	Options that control the use of the file. Possible values are: <ul style="list-style-type: none">■ TPE_FILE_CREATE – Creates a new file. An error occurs if the file exists.■ TPE_FILE_OVERWRITE – Replaces any existing file.■ TPE_FILE_APPEND – Appends fetched data to an existing file or creates a new file.
name	The name of the file.

**CLOB_
LOCATOR**

The CLOB_LOCATOR data type defines a locator variable for a CLOB value. An application may use a locator variable to manipulate a CLOB value at StorHouse.

CLOB_LOCATOR specification

C language representation	int32_t
Declare CLOB locator variables as	CLOB_LOCATOR
Example	EXEC SQL BEGIN DECLARE SECTION; CLOB_LOCATOR part_text; EXEC SQL END DECLARE SECTION;

DATE

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

DATE specification

C language representation	tpe_date_t
Declare DATE variables as	date or DATE
Example	EXEC SQL BEGIN DECLARE SECTION ; date date_v ; EXEC SQL END DECLARE SECTION ;

2

Satisfying program requirements

Declaring variables and types

The structure definition for tpe_date_t is:

```
typedef struct {
    uint16_t    year;
    uint8_t     month;
    uint8_t     day;
} tpe_date_t;
```

Field	Description
year	The year part of the date represented as a short data type ranging in value from 1 to 9999.
month	The month part of the date represented as an unsigned char data type ranging in value from 1 to 12.
day	The day part of the date represented as an unsigned char data type ranging in value from 1 to the upper limit for the specific month and year.

DOUBLE
(synonym FLOAT)

The DOUBLE data type defines data as a double precision floating-point number.

DOUBLE or FLOAT specification	
C language representation	double
Declare DOUBLE variables as	double, DOUBLE, float, or FLOAT
Example	EXEC SQL BEGIN DECLARE SECTION ; double salary_v ; EXEC SQL END DECLARE SECTION ;

INTEGER The INTEGER data type represents data as an integer with a 4-byte length.

INTEGER specification

C language representation	int32_t
Declare INTEGER variables as	INTEGER or int32_t
Range of values	-2147483648 to +2147483647
Example	EXEC SQL BEGIN DECLARE SECTION ; long qty_v ; EXEC SQL END DECLARE SECTION ;

NUMERIC
(or **DECIMAL**) The NUMERIC data type (synonym DECIMAL) represents data as a number with a given precision and scale.

NUMERIC or DECIMAL specification

C language representation	tpe_num_t The value for TPE_NUMERIC_SIZE in the tpe_num_t structure definition is 16. The size of the entire tpe_num_t structure is 24 bytes, which is the space your program must allow and is the size shown by the SQLDA.
Declare NUMERIC variables as	numeric, NUMERIC, decimal, or DECIMAL
Restrictions	When you declare a variable using the NUMERIC type, you cannot specify precision and scale.
Example	EXEC SQL BEGIN DECLARE SECTION ; numeric commission_v ; EXEC SQL END DECLARE SECTION ;

The structure definition for `tpe_num_t` is:

```
typedef struct {
    int16_t    precision;
    int16_t    scale;
    int16_t    exp;
    int16_t    dec_num;
    char       dec_digits[TPE_NUMERIC_SIZE];
} tpe_num_t;
```

Field	Description
precision	The total number of decimal digits in the number.
scale	The number of decimal digits to the right of the decimal point.
exp	Reserved for future use and should be 0.
dec_num	The number of bytes used in the dec_digits field.
dec_digits	The numeric data in binary coded decimal (BCD) digits.

NUMERIC values are represented as binary coded decimal (BCD), with two decimal digits per byte (4 bits per digit.) The last (rightmost) digit position contains the sign of the value. The hexadecimal value C (bit pattern 1100) indicates a positive number and D (1101) represents a negative number. If the precision of a NUMERIC value is even, an extra digit is added on the left to maintain alignment. For example, below are some values and their in-memory representations:

Precision and scale	Value	Representation (hexadecimal)
(5, 2)	134.25	13 42 5c
(8, 3)	-27125.21	02 71 25 21 0d
(10, 2)	1935.22	00 00 01 93 52 2c

In the `tpe_num_t` data structure, StorHouse may eliminate leading zero bytes in the value. This means that the precision of the value in the `dec_digits` field may

be less than the precision indicated in the precision field. This effective precision is calculated as follows:

$$(\text{dec_num} * 2) - 1$$

For instance, the value 1935.22 listed in the preceding table has the following values in a tpe_num_t structure. The effective precision is 7.

Field	Value
precision	10
scale	2
exp	0
dec_num	4
dec_digits	01 93 52 2c

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

REAL specification	
C language representation	float
Declare REAL variables as	REAL or real
Example	EXEC SQL BEGIN DECLARE SECTION ; real val ; EXEC SQL END DECLARE SECTION ;

2

Satisfying program requirements

Declaring variables and types

SMALLINT The SMALLINT data type defines data as a signed small integer with a 2-byte length.

SMALLINT specification

C language representation	int16_t
Declare SMALLINT variables as	SMALLINT, short, or int16_t
Range of values	-32768 to +32767
Example	EXEC SQL BEGIN DECLARE SECTION ; short deptno_v ; EXEC SQL END DECLARE SECTION ;

TIME The TIME data type represents data as a four-part time value: hours, minutes, seconds, and milliseconds.

TIME specification

C language representation	tpe_time_t
Declare TIME variables as	time or TIME
Example	EXEC SQL BEGIN DECLARE SECTION ; time time_v ; EXEC SQL END DECLARE SECTION ;

The structure definition for tpe_time_t is:

```
typedef struct {
    uint8_t    hours;
    uint8_t    mins;
    uint8_t    secs;
    uint8_t    reserved;
    uint16_t   msecs;
} tpe_time_t;
```

Field	Description
hours	The hours portion of the time represented as an unsigned char data type ranging in value from 0 to 24. If hours equal 24, then mins, secs, and msecs must equal 00.
mins	The minutes portion of the time represented as an unsigned char data type ranging in value from 0 to 59.
secs	The seconds portion of the time represented as an unsigned char data type ranging in value from 0 to 62. You can use seconds greater than 60 to indicate leap seconds. However, time calculations do not consider leap seconds.
reserved	Reserved for future use.
msecs	The milliseconds portion of the time represented as an unsigned short data type ranging in value from 0 to 999.

TIMESTAMP The StorHouse TIMESTAMP data type represents data as a date and time in a 7-part format.

TIMESTAMP specification	
C language representation	tpe_timestamp_t
Declare TIMESTAMP variables as	timestamp or TIMESTAMP
Example	EXEC SQL BEGIN DECLARE SECTION ; timestamp birth_info; EXEC SQL END DECLARE SECTION ;

The structure definition for `tpe_timestamp_t` is:

```
typedef struct {
    uint16_t    year;
    uint8_t     month;
    uint8_t     day;
    uint8_t     hours;
    uint8_t     mins;
    uint8_t     secs;
    uint8_t     reserved;
    uint32_t    usecs;
} tpe_timestamp_t ;
```

Field	Description
year	The year part of the date represented as a short data type ranging in value from 1 to 9999.
month	The month part of the date represented as an unsigned char data type ranging in value from 1 to 12.
day	The day part of the date represented as an unsigned char data type ranging in value from 1 to the upper limit for the specific month and year.
hours	The hours portion of the time represented as an unsigned char data type ranging in value from 0 to 23. If hours equal 24, then mins, secs, and usecs must equal 00.
mins	The minutes portion of the time represented as an unsigned char data type ranging in value from 0 to 59.
secs	The seconds portion of the time represented as an unsigned char data type ranging in value from 0 to 62. You can use seconds greater than 60 to indicate leap seconds. However, time calculations do not consider leap seconds.
usecs	The microseconds portion of the time represented as an unsigned long data type with values ranging from 0 to 999999.

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

VARBINARY specification

C language representation

unsigned char

Restrictions

- *Never* declare host variables as the VARBINARY data type. Instead, use an SQLDA and the DESCRIBE statement. See Chapter 5 for more information about SQLDAs and DESCRIBE.
- When StorHouse/RM returns a variable-length binary value, the first two bytes contain the actual length of the data. You should process these two bytes as a short integer. Move (for example, memcpy) these first two bytes to a short variable because the returned field may not be aligned.
- *Never* handle VARBINARY data as a null-terminated string because the data in a VARBINARY field may contain 0 (NUL) bytes.

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

VARCHAR specification

C language representation

char

Restrictions

- *Never* declare host variables as the VARCHAR data type. Instead, use an SQLDA and the DESCRIBE statement. See Chapter 5 for more information about SQLDAs and DESCRIBE.
- When StorHouse/RM returns a variable-length character value, the first two bytes are the actual data length. These two bytes should be processed as a short integer. Move (for example, memcpy) these two bytes to a short variable because the field may not be aligned.

Declaring variables with type definitions

You can define a new data type (or alias) with the same characteristics as an existing host or StorHouse data type. Then you can declare host variables using the new data type. You define new types with the TYPE IS OF TYPE statement in a Declare Section.

The format of TYPE IS OF TYPE is:

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 [unsigned] long [unsigned] short double In place of C type short or long, you can use a guaranteed-size type listed in the table on page 2-4.
storhouse_type	(required for specifying a StorHouse 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

The following example declares the new type, `customer_no`, with the characteristics of the C type `long`. It also declares the host variable `input_v` as the type `customer_no`:

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

Declaring variables as host arrays

You can process a collection of data elements with one SQL statement by using arrays. An *array* is a group of data items, or elements, assigned to one variable name. You assign a host array to a host variable name and an indicator variable array to an indicator variable name. You associate an indicator variable array with a host variable array.

ESQL maps all arrays into the host language structure `new_type_name`. This structure consists of the actual array and the current size of that array.

The C language definition of the `new_type_name` structure is:

```
struct new_type_name {
    long tpe_size;
    element_type_name tpe_array[constant_id];
};
```

Argument	Definition
<code>tpe_size</code>	The current size of the array.
<code>tpe_array[constant_id]</code>	The actual array. <code>Constant_id</code> indicates the number of elements in the array of type <code>element_type_name</code> .

You can use C statements to process an array as long as you define the array as a structure with the same name as the array name and with the two components `tpe_array` and `tpe_size`. In addition, you must manually update `tpe_size` to the

number of rows that are returned. In contrast, the SQL statement `FETCH` updates `tpc_size` automatically. See Chapter 3, “Submitting queries in ESQL,” for an explanation of `FETCH`.

You use the `TYPE` statement to declare and set the size of host arrays in a `Declare` Section. There are three ways to declare an array with `TYPE`:

- In ESQL syntax
- In C syntax
- As a new data type, then declare the host array variable using the new data type

ESQL format

The ESQL format for declaring a host array is:

`variable_name IS AN ARRAY OF type_name WITH SIZE constant_id`

Argument	Description
<code>variable_name</code>	(required) The name of the array being declared.
<code>type_name</code>	(required) The data type assigned to <code>variable_name</code> .
<code>constant_id</code>	(required) The number of elements in <code>variable_name</code> .

The following example uses the ESQL format to declare the host array `my_array` of C type `long`. There are 12 elements in the array:

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

C format

The C format for declaring a non-char host array is:

host_language_type_name variable_name [constant_id]

Argument	Description
host_language_type_name	(required) The data type assigned to variable_name.
variable_name	(required) The name of the array being declared.
constant_id	(required) The number of elements in variable_name.

The following example uses the C format to declare a host array of type long. There are 12 elements in the array:

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

The corresponding C language definition is:

```
struct my_array{
    long tpe_size;
    long tpe_array [12];
}
```

The C format for declaring a char host array is:

host_language_type_name variable_name [constant_id] [length]

Argument	Description
host_language_type_name	(required) The data type assigned to variable_name.
variable_name	(required) The name of the array being declared.

2

Satisfying program requirements

Declaring variables and types

Argument	Description
constant_id	(required) The number of elements in variable_name.
length	(required) The length of each char element in variable_name.

The following example uses the C format to declare the array my_array of C type char. There are 12 elements in the array. Each element is five characters long:

```
EXEC SQL BEGIN DECLARE SECTION ;
      char my_array [12][5] ;
EXEC SQL END DECLARE SECTION ;
```

The corresponding C language definition is:

```
struct my_array {
    long tpe_size:
    char tpe_array [12] [5];
}
```

Declaring an array as a new data type

The format for declaring an array as a new data type is:

```
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) The host language data type or StorHouse data type that appears in the array.
constant_id	(required) The size of each element in the array.

The following example declares a new type called `array_type` with the same characteristics as the C data type `char`. This data type represents an array where each element contains 20 characters:

```
EXEC SQL BEGIN DECLARE SECTION ;
    TYPE array_type IS AN ARRAY OF char
    WITH SIZE 20;
EXEC SQL END DECLARE SECTION;
```

The following example uses ESQL syntax to declare the host array `my_array` of type `array_type`. There are 30 elements in the array:

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

Using host variables

You can use host variables in your ESQL program for input and output operations. For example, you can use:

- One or more output host variables in a `SELECT` or `FETCH` statement `INTO` clause to hold values returned by StorHouse/RM
- A locator variable with a `VALUES INTO` statement to manipulate a LOB value at the StorHouse server
- An output file reference variable to transfer data from StorHouse to a client file

Before StorHouse/RM executes any SQL statements that contain input host variables, your program must assign values to them. You may not use a host variable to represent a table, view, or column name.

Locator variables and file reference variables follow the basic guidelines in this section. See Chapter 6, “Accessing large objects” for specific information and examples for using these types of host variables.

The format for specifying a host variable in an ESQL statement is:

`:host_variable_name`

When you use a host variable, you must:

- Explicitly declare the host variable in a Declare Section
- Declare the host variable before it is used in an ESQL program
- Always precede the host variable by a colon (:) in an ESQL statement
- Use the host variable in an ESQL statement only where you can use a constant

You must not:

- Use host variables in CREATE, ALTER, and DROP statements
- Precede a host variable by a colon when you use it in C language statements
- Use SQL reserved words as host variable names

The following example uses the output host variables `:emp_num_v` and `:dept_id_v` in a SELECT statement INTO clause. You must have previously declared these host variables in a Declare Section as explained at “Coding a Declare Section” on page 2-1.

```
EXEC SQL
  SELECT emp_num, dept_id
  INTO :emp_num_v, :dept_id_v
  FROM emptable
  WHERE emp_num = 3904933 ;
```

Using indicator variables

You can use StorHouse indicator variables to test for null or truncated values. You always associate an output indicator variable with an output host variable.

The format of an indicator variable in an ESQL statement is:

:host_variable_name :indicator_variable_name

When using indicator variables, you must:

- Explicitly declare them in a Declare Section
- Declare them as a short or SMALLINT data type
- Precede them with a colon in ESQL statements
- Precede them by their associated host variables in ESQL statements, separated by a space

You must not:

- Precede indicator variables with a colon in C language statements
- Use SQL reserved words as indicator variable names
- Use indicator variables in WHERE clauses

Valid values for indicator variables are:

Value	Indicates that
0	A non-null value was placed in the host variable, and the host variable was not truncated.
-1	The returned value is null and no value has been returned in the host variable.
>0	The returned value was truncated because the host variable size was too small. For non-LOB types, the indicator variable value is the actual length of the returned value before truncation. For LOB types, the indicator variable value is 1.

For locator variables, the meaning of the indicator variable values is slightly different. Because a locator variable cannot be NULL, a negative indicator variable value indicates that the LOB value represented by the locator is NULL. The NULL information is kept local to the client using the indicator variable value. StorHouse/RM does not track NULL values with locator variables.

You can determine whether a returned value is NULL or truncated by using an output host variable and its associated indicator variable in a SELECT statement INTO clause. The following example checks the value of the indicator variable `projno_i` to determine if the output host variable `projno_v` contains a NULL value. If `projno_i` contains a -1, then `projno_v` contains a NULL value.

```
EXEC SQL
    SELECT ename, job, deptno, projno
    INTO :ename_v, :job_v, :deptno_v, :projno_v :projno_i
    FROM employee
    WHERE empron=1250;
if(projno_i== -1)
    printf ("project number is NULL\n");
...
```

Managing connectivity to StorHouse

Your program must connect to a StorHouse database before it can perform database system administration tasks or submit queries. System-wide, the `SQL_SESSIONS` system parameters controls the maximum number of connections. A single program can connect to up to 10 databases at a time; however, you can execute SQL statements in only one database at a time. The database where you execute SQL is your *current connection*.

The StorHouse SQL statements that help you manage database connectivity are:

- CONNECT
- SET CONNECTION
- DISCONNECT

Connecting to a StorHouse database

You connect to a StorHouse database by issuing the SQL CONNECT statement and specifying the:

- Name of the database you want to access expressed as a remote database name
- StorHouse account ID and password to validate your access to StorHouse
- Name you're assigning to this connection

Caution: The StorHouse account password is optional (in the USING clause of CONNECT); but, if you omit it, StorHouse will prompt stdin for the password. If stdin is not a terminal, the CONNECT will fail.

CONNECT must be the first executable SQL statement in your program. Only host language code and declarative SQL statements may logically appear before CONNECT.

The format of the CONNECT statement is:

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

Argument	Description
database_name	(required) Name of the StorHouse 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.

Argument	Description
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.

The following example specifies a connection string for the StorHouse remote database salesdb. The name assigned to this connection is conn_2. The program uses the account ID ssc and the account password represented by the host variable :sscpswd to validate user access to StorHouse.

```
EXEC SQL
    CONNECT TO 'filetek:T:remotehost:salesdb' AS 'conn_2'
    USER 'sscl' USING :sscpswd ;
```

The components of the connection string filetek:T:remotehost:salesdb are:

- filetek:T is a constant
- remotehost is the name of the remote machine containing salesdb
- salesdb is the name of the remote database
- conn_2 is the connection name

Changing the current connection

The SET CONNECTION statement resumes the connection associated with the specified connection name and makes that connection the current one. It restores the context of the current database to the same state that prevailed when the connection was previously suspended.

The format of SET CONNECTION is:

```
EXEC SQL
  SET CONNECTION connection_name ;
```

Argument	Description
connection_name	(required) The name of the connection you are restoring. This connection must have been established by a previous CONNECT statement and must not have been terminated by a previous DISCONNECT statement.

The following example sets the current connection to the database represented by the conn_1 connection string:

```
EXEC SQL
  SET CONNECTION 'conn_1' ;
```

Terminating a connection

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

The format for DISCONNECT is:

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

2**Satisfying program requirements**

Managing connectivity to StorHouse

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.

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

```
EXEC SQL
    DISCONNECT 'conn_2' ;
```

Submitting queries in ESQL

This chapter explains how to submit queries in embedded SQL. It describes:

- The SELECT statement
- Cursors
- Array fetches

About queries

A *query* is a database operation that retrieves relational data from tables. Queries can return one or more rows depending on your selection criteria. You submit queries in ESQL with the SELECT statement. The column names and expressions that follow the keyword SELECT make up your *select list*. This list identifies the information you want to retrieve.

In the following query, the select list contains the columns name and city:

```
EXEC SQL
  SELECT name, city
  FROM customer ;
```

The SELECT statement supports the following clauses:

SELECT clause	Provides the
FROM	Name of the table(s) you want to access and any join specifications
GROUP BY	Criteria for grouping rows returned by your query
HAVING	Criteria for applying one or more qualifying conditions for selected groups
INTO	Variables used to contain a one-row result set
ORDER BY	Order of the rows selected by the query
WHERE	Search condition(s) to be used for row selection
FOR	Compatible clause to allow IBM® DB2™ programs to work correctly without changing the SQL

You use input host variables in WHERE and HAVING clauses and output host variables in the INTO clause.

Queries that return a single row

You use the SELECT statement INTO clause for queries that return only one row. The number of output host variables in the INTO clause must always equal the number of columns in the select list. If you use SELECT INTO and your query returns more than one row, the query fails.

If your query accesses a LOB value, you can place the value into a host variable that is large enough to hold it. That is, the SELECT statement INTO clause host variable may be defined with a BLOB or CLOB data type. The INTO clause host variable may also be a locator variable or a file reference variable. See Chapter 6, “Accessing large objects,” for more information about defining host variables for LOB data.

The following example declares two output host variables: `name_v` and `city_v`. Then it requests name and city information for a specific customer number from

the customer table. You can assume that the customer table has no duplicate entries, so this query will return only one row. The query specifies customer number 1234 in the WHERE clause.

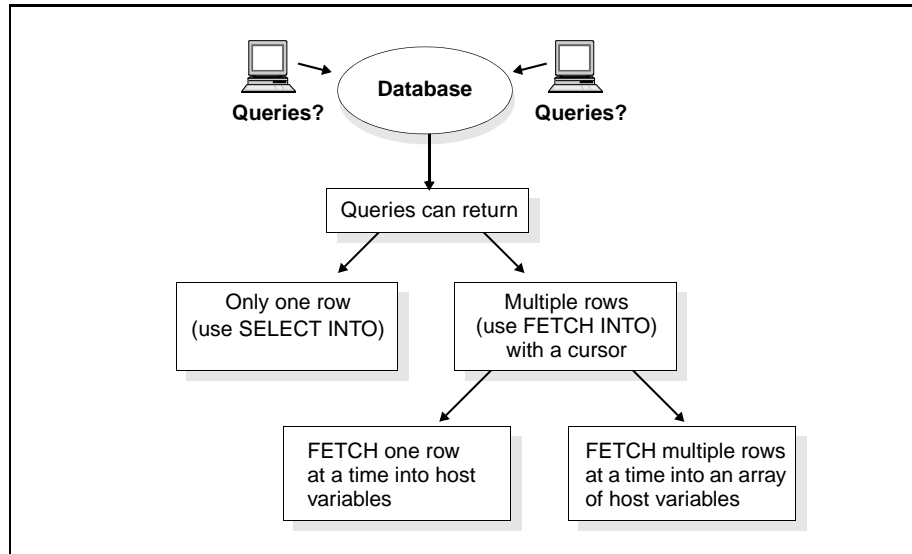
```
EXEC SQL BEGIN DECLARE SECTION ;
    char name_v [30] ;
    char city_v [20];
EXEC SQL END DECLARE SECTION ;
...
EXEC SQL
    SELECT name, city
    INTO :name_v, :city_v
    FROM customer
    WHERE cust_no = 1234 ;
```

Based on the selection criteria specified in the WHERE clause, the query returns values for customer name and city into the output host variables, name_v and city_v.

Queries that return multiple rows

For queries that return more than one row, you use a cursor to retrieve, or *fetch*, one row at a time or an array of rows into output host variables. See page 3-7 for an explanation of FETCH. The following graphic summarizes how to retrieve one

or more rows with a query.



You can use cursor and fetch operations to access LOB data. See Chapter 6, “Accessing large objects,” for more information about using the FETCH statement INTO clause to access LOB data.

Using cursors

You must explicitly define a cursor for queries that can return more than one row. StorHouse/RM uses a cursor to retrieve each row in the result set and to indicate the current row.

You use the following SQL statements to control cursors:

SQL statement	Definition
DECLARE	Names a cursor and associates it with a query.
OPEN	Executes the query and identifies the result set.

SQL statement	Definition
FETCH	Retrieves one row or an array of rows from the result set. See page 3-9 for a discussion of array fetches.
CLOSE	Terminates cursor processing and releases system resources.

A cursor can be in one of two states: open or closed. An *open cursor* is associated with an active set. A *closed cursor* is no longer associated with an active set, although it remains associated with the SELECT statement.

The following rules apply to cursors:

- Cursor names must begin with a letter. They can consist of any combination of letters a-z and A-Z, numbers 0-9, and underscore (_). Cursor names should be no longer than 32 characters.
- You must declare a cursor before you reference it in any SQL statement. ESQL cannot interpret a reference to an undeclared cursor.
- All cursor control statements must occur within the same precompiled unit. In other words, you cannot declare a cursor in one file and open it in another.
- Cursor names must be unique within a file.

Associating a cursor with a query

You use the DECLARE statement to assign a name to a cursor and to associate the cursor with a specific query (SELECT statement). The cursor name is an identifier for the precompiler. It's not a host variable, so it should not be defined in a Declare Section.

The format for DECLARE is:

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

Use the `query_expression` option with embedded SQL and the `prepared_statement_name` option with dynamic SQL on the PREPARE statement. See page 5-7 for information about PREPARE.

The following example declares the cursor `cust_cur` for a specific SELECT statement (`query_expression`):

```
EXEC SQL DECLARE cust_cur CURSOR FOR
  SELECT name, city, state
  FROM customer
  WHERE cust_no = 1234 ;
```

Opening a cursor

The OPEN statement opens a cursor, executes the associated SELECT statement with the current program variables, and identifies the result set.

The format for OPEN is:

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

Use the USING clause with embedded SQL and the USING DESCRIPTOR clause with dynamic SQL. See page 5-12 for information about USING DESCRIPTOR.

ESQL references input host variable values when you open a cursor. This means that changing the value of input host variables after OPEN does not affect your result set. You must close the cursor and then reopen it for changes to affect your

result set.

The following example declares and opens the cursor `cust_cur`:

```
EXEC SQL
  DECLARE cust_cur CURSOR FOR
  SELECT name, city, state
  FROM customer
  WHERE cust_no = 1234;
EXEC SQL OPEN cust_cur ;
```

Retrieving rows using a cursor

The `FETCH` statement reads the rows of the result set and returns the values into host variables. For queries that can return multiple rows, `FETCH`, rather than `SELECT`, contains the `INTO` clause with the list of output host variables. The number of output host variables in the `INTO` clause must always equal the number of items in the select list.

The format for `FETCH` is:

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

Use the `INTO` clause with embedded SQL and the `USING DESCRIPTOR` clause with dynamic SQL. See page 5-12 for information about `USING DESCRIPTOR`.

The following code fetches rows opened by the cursor `cust_cur`:

```
/* connect to your database */
...

EXEC SQL
    DECLARE cust_cur CURSOR FOR
    SELECT name, city, state
    FROM customer
    WHERE cust_no = 1234;

/* Open cursor */

EXEC SQL OPEN cust_cur;

/* Fetch the query results into host variables */

while (sqlca.sqlcode==0)
{
    EXEC SQL
        FETCH cust_cur
        INTO :name_v, :city_v; :state_v;
}
EXEC SQL
    CLOSE cust_cur;
...
```

The preceding example declares and opens the cursor `cust_cur`. The first execution of `FETCH` retrieves the first row of the active set. This row becomes the current row. The cursor can only move forward in the active set. Each subsequent `FETCH` advances the cursor to the next row. The only way to return to a previously fetched row is to close the cursor and reopen it.

If the cursor is already pointing to the last row of the active set or if the active set does not contain any rows, a `FETCH` returns the status code 100 in the `SQLCA` field `sqlcode` to indicate that there are no more rows to be fetched. In this case, you must close and reopen the cursor before you can use it again.

Closing a cursor

The CLOSE cursor statement closes a cursor. Once you close a cursor, you can't use it to FETCH a row because you no longer have an active set. No statements referring to the cursor, except for OPEN, are operative.

The format for CLOSE is:

```
EXEC SQL
    CLOSE cursor_name ;
```

The following example closes the cursor cust_cur:

```
EXEC SQL
    CLOSE CURSOR cust_cur ;
```

Using a cursor with host variable arrays

The FETCH statement typically returns one row at a time from the active set selected by OPEN. ESQL can fetch multiple rows at a time if you declare your host variables as arrays. Array fetching is more efficient than single row fetches for retrieving a large number of rows because it reduces the number of calls to a database.

Note: StorHouse/RM currently does not support array fetches with file reference variables (BLOB_FILE and CLOB_FILE data types).

StorHouse/RM maps all host arrays to a C structure, which is defined on page 2-21. The structure element tpe_size initially contains the maximum number of rows that an application can retrieve with a single FETCH call. In a multi-row fetch, you must reset tpe_size before every FETCH.

For example, if `tpe_size` is initially equal to 25 and there are 60 rows in your result set, you'll fetch 25 rows with the first `FETCH`, 25 rows with the second `FETCH`, and 10 rows with the third `FETCH`. You must reset `tpe_size` as shown in the following table.

Operation	Set <code>tpe_size</code> to
First <code>FETCH</code>	25
Second <code>FETCH</code>	25
Third <code>FETCH</code>	10

The third fetch also sets `sqlcode` to 100, which indicates that there are no more rows to be retrieved. In this case, `FETCH` actually retrieved 10 rows even though the `sqlcode` indicates the `SQL_NOT_FOUND` condition.

Note: An extractor cursor operation returns the partial buffer with `sqlcode` 0 and an empty buffer with `sqlcode` 100.

The following example declares the output host variable arrays `name` and `cust_no`. It uses these arrays to fetch up to 50 rows in one `FETCH` call:

```
/* Fetch up to 50 rows in one fetch call */
#define ARRAYSZ 50
#define NAMESZ 30

EXEC SQL BEGIN DECLARE SECTION;

TYPE customer_name_t IS AN ARRAY OF CHAR WITH SIZE NAMESZ;
TYPE customer_id_t IS OF TYPE LONG INTEGER ;

customer_name_array IS AN ARRAY OF customer_name_t
    WITH SIZE ARRAYSZ;
customer_id_array IS AN ARRAY OF customer_id_t
    WITH SIZE ARRAYSZ;

EXEC SQL END DECLARE SECTION;

...
```

```
EXEC SQL
DECLARE customer_cursor CURSOR FOR
  SELECT name, cust_no
  FROM customer ;
...
if (sqlca.sqlcode < 0) goto err ;
```

3

Submitting queries in ESQL

Using a cursor with host variable arrays

Handling errors and warnings

You've already learned how to use indicator variables to check for NULL or truncated values. This chapter describes other ways to handle error and warning conditions. These include using the:

- SQLCA
- WHENEVER statement

Using the SQLCA

ESQL programs require a data structure called the SQL Communications Area (SQLCA) to hold information about the status of your most recently executed SQL statement. StorHouse/RM updates the SQLCA after every executable SQL statement.

You can use the SQLCA to check:

- Return code (sqlcode) information
- The number of rows fetched
- Warning flags (sqlwarn)

C programs implement the SQLCA as a global structure that is automatically declared and defined by the ESQL precompiler.

SQLCA structure definition

The components of the SQLCA structure are:

```
struct tpe_sqlca {
    char      sqlcaid[8];          /* Eye-catcher, "TPESQLCA" */
    int32_t   sqlcode;             /* Result of execution */
    int16_t   sqlcabc;             /* Length of tpe_sqlca */
    uint16_t  sqlerrml;            /* Length of message */
    char      sqlerrm[74];         /* Null-terminated message */
    char      sqlerrp[8];          /* (reserved) */
    int32_t   sqlerrd[8];          /* Diagnostic information */
    char      sqlwarn[9];          /* Warning flags */
    char      sqlstate[5];         /* Completion state */
};
```

Component	Description
sqlcaid	Contains the string 'SQLCA'.
sqlcode	<p>Provides the result of an SQL statement execution. Valid values are:</p> <ul style="list-style-type: none"> ■ 0 – indicates successful execution. ■ positive – indicates successful execution. The only positive status code is 100 (SQL_NOT_FOUND), which is returned when there are no more rows to be fetched. ■ negative – indicates that an error occurred in SQL statement execution. Refer to the <i>StorHouse SQL Reference Manual</i> for a list of SQL codes.
sqlcabc	Contains the size of the SQLCA structure.
sqlerrml	Contains the length of the error message in sqlerrm.
sqlerrm	Contains a null terminated character string that is the error text corresponding to the code in sqlcode.
sqlerrp	Currently not used.

Component	Description
sqlerrd	Contains an array of six long integers. Elements 0, 1, and 4 are currently not used.
sqlerrd[2]	Number of rows that were processed after a successful execution of INSERT, UPDATE, and DELETE (for system tables and system table views only). For FETCH, this element is set to the cumulative number of rows for a FETCH call associated with a cursor.
sqlerrd[3]	Number of rows fetched by the last FETCH call.
sqlerrd[5]	Value indicating whether StorHouse/RM used the extractor or the database engine to process a query. This value is set only after a PREPARE. Valid values are: <ul style="list-style-type: none"> ■ Non-zero – the extractor. ■ 0 – the database engine.
sqlwarn	Contains a character array with eight elements. Currently StorHouse/RM uses only elements 0, 1, 2, 3, 4, and 6. Valid values are blank or the single character 'W,' which indicates a warning during SQL statement execution.
sqlwarn[0]	If set to W, then one or more other elements in this array are also set to W. If blank, then no warnings are set.
sqlwarn[1]	If set to W, then one or more strings returned by the previous FETCH were truncated. Use indicator variables to indicate which strings were truncated.
sqlwarn[2]	If set to W, then one or more NULL values were ignored in the computation of an aggregate function.
sqlwarn[3]	If set to W, then the number of items in the SELECT list does not equal the number of host variables in the INTO clause. The query returns n elements, where n is the lesser of the two.
sqlwarn[4]	If set to W, then an UPDATE or DELETE without a WHERE clause completed successfully. It's useful to check this element when working with dynamic statements. UPDATE and DELETE are valid for system tables and system table views only.
sqlwarn[6]	If set to W, then the transaction is implicitly marked for ROLLBACK. The application must roll back the current transaction before executing the next SQL statement.
sqlstate	Is reserved for future use.

Checking the sqlcode for status codes

You can use the SQLCA component `sqlcode` to check status codes. You can use other components to supply additional diagnostic information and to check for warnings.

The following example shows how to check for the value 100 in `sqlcode` while using `FETCH`:

```
...
/* Open cursor */
EXEC SQL OPEN cust_cur ;

if (sqlca.sqlcode !=0)
{
    fprintf (stderr,
        "Open cursor statement failed (%ld : %s)\n",
        sqlca.sqlcode, sqlca.sqlerrm);
    return (-1);
}

/* Fetch rows and return result set into host variables */
while (sqlca.sqlcode==0)
{
    EXEC SQL FETCH cust_cur
        INTO :cust_name_v, :cust_city_v,:cust_state_v;

    if (sqlca.sqlcode==0)
    {
        printf ("cust_no : %d, name : %s, city : %s\n",
            cust_no_v, name_v, city_v) ;
    }
}
if (sqlca.sqlcode < 0)
{
    fprintf (stderr,
        "FETCH cursor statement failed (%ld : %s)\n",
        sqlca.sqlcode, sqlca.sqlerrm);
}
```

```
EXEC SQL CLOSE cust_cur ;
EXEC SQL DISCONNECT 'conn_1' ;
...
```

Checking for warnings

You can use the SQLCA component sqlwarn to check for warnings that occur during the execution of an ESQL statement.

The following example shows how to use sqlwarn. It computes the average commission for employees in the sales department and checks the sqlwarn[2] component to determine whether any NULL values were ignored in the computation of the average commission.

```
EXEC SQL BEGIN DECLARE SECTION ;
      FLOAT comm_v ;
EXEC SQL END DECLARE SECTION ;
...
EXEC SQL
      SELECT AVG (commission)
      INTO :comm_v
      FROM employee
      WHERE deptno = 20 ;
if (sqlcode==0)
{
/* Check for SQLCA warnings */
  if (sqlca.sqlwarn[2]=='W')
  {
    printf ("(One or more NULL values ignored\n") ;
    printf (" in computation of average commission !)\n") ;
  }

  printf ("commission : %d\n", comm_v) ;
}
...
```

Using WHENEVER

By default, precompiled programs ignore error and warning conditions and continue processing when possible. StorHouse SQL supports the WHENEVER statement to automate condition checking and error handling for you.

WHENEVER checks the SQLCA for three runtime exceptions:

- SQLERROR
- SQLWARNING
- NOT FOUND

Depending on the exception, you can tell your program to continue with the next statement, branch to a host language label, or stop execution.

The format of WHENEVER is:

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	One of the following 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	One of the following specific actions:
STOP	Terminate program without any final reporting.
CONTINUE	Ignore the specified exception and continue executing the next program statement. CONTINUE is the default.
GOTO host_language_label	Branch to the statement corresponding to the host_language_label.

Note: FileTek recommends that you use CONTINUE or GOTO instead of STOP.

Do not use WHENEVER NOT FOUND with array FETCH. If you are retrieving multiple rows at a time (see the section, “Using a cursor with host variable arrays,” on page 3-9), FETCH may return rows from the result set and still specify 100 (no more rows to be found) in the sqlca.

Scope of WHENEVER

The scope of WHENEVER is positional rather than logical within a program. *Positional* means that WHENEVER tests all executable SQL statements that physically follow it in the source file, not in the flow of program logic. You should always code WHENEVER before the first executable SQL statement you want to test. A WHENEVER statement stays in effect until it is superseded by another WHENEVER statement that checks for the same condition or until the end of the source file.

In the following example, the second WHENEVER SQLERROR statement supersedes the first. In other words, the first WHENEVER only applies to the CONNECT statement. The second WHENEVER SQLERROR statement applies to both the DELETE and DROP statements, in spite of the control flow from step1 to step3.

```
step1:
    EXEC SQL WHENEVER SQLERROR GOTO step4;
    EXEC SQL CONNECT.....
    ...
    GOTO step3
step2:
    EXEC SQL WHENEVER SQLERROR CONTINUE
    EXEC SQL DELETE sysadm.syssmusers WHERE accountid='act1';
    ...
step3:
    EXEC SQL DROP TABLE mytable;
    ...
step4:
    ...
```

The following example uses the WHENEVER statement for exception handling. In this example, the CONTINUE in the second WHENEVER statement prevents control from passing to the do_rollback label, which prevents the program from going into an endless loop if there are subsequent errors.

```
...
/* Upon error branch to label do_rollback */
EXEC SQL WHENEVER SQLERROR GOTO do_rollback ;

EXEC SQL
    CREATE TABLE newtable
        (COL1DEF CHAR (5))
        TABLE SPACE CV7SET2;
```



```
/* Commit work and disconnect from database */
EXEC SQL COMMIT WORK ;
EXEC SQL DISCONNECT filetek:T:sys1:cust_db ;
exit (0) ;
do_rollback:
EXEC SQL WHENEVER SQLERROR CONTINUE ;

strncpy (errmsg, sqlca.sqlerrm, sqlca.sqlerrml);
errmsg [sqlca.sqlerrml] = '\0' ;
fprintf (stderr, "Error : %s\n", errmsg);

EXEC SQL ROLLBACK WORK ;
EXEC SQL DISCONNECT filetek:T:sys1:cust_db ;
exit (1) ;
```

It's a good practice to code the following statement:

```
EXEC SQL WHENEVER <condition> CONTINUE;
```

at the end of any function that contains a WHENEVER...GOTO declaration.

4

Handling errors and warnings

Using WHENEVER

Using dynamic SQL

This chapter defines dynamic SQL and explains:

- When to use dynamic SQL
- How to represent dynamic SQL statements as character strings
- How to use substitution markers for input host variables in dynamic SQL
- Four scenarios for accepting and processing SQL statements at runtime
- How to choose the appropriate scenario for your job
- How to use dynamic SQL to fetch from host arrays

About dynamic SQL

Some applications accept and process different SQL statements at every program execution. These applications may submit queries that select a variety of information from different tables according to search criteria received at runtime. Such SQL statements are *dynamic* because they vary from one program execution to the next. Dynamic SQL statements provide more application flexibility than embedded SQL because they allow for change. You need this versatility when you don't know one or more of the following at precompile time:

- The specific SQL statements you want to execute
- Number of host variables in each statement
- Data types of host variables
- Tables, columns, or view names

Storing dynamic SQL as a character string

Unlike static SQL, dynamic SQL statements are not embedded in your program. Instead, they're stored as character strings. You can declare them as character host variables in a Declare Section or represent them as quoted string literals. Either way, these character strings can be input to or constructed by your program at runtime.

The following example illustrates how to store a dynamic SQL statement as a host variable. This example declares the host variable `sql_str` in a Declare Section and assigns it the value "select name, id from customer_table". StorHouse/RM can use this value to build a dynamic SQL statement.

```
EXEC SQL BEGIN DECLARE SECTION ;
      char sql_str [256] ;
EXEC SQL END DECLARE SECTION ;
strcpy (sql_str,"select name, id from customer_table");
```

Character strings that represent dynamic SQL statements may not contain:

- The keywords EXEC SQL
- The statement terminator for C (the semicolon)
- Any of the following SQL statements
 - CLOSE
 - DECLARE
 - DESCRIBE
 - EXECUTE
 - FETCH
 - INCLUDE
 - OPEN
 - PREPARE
 - WHENEVER

Understanding substitution markers

Dynamic SQL may contain *place holders*, or *substitution markers*, for host variables that are substituted in an SQL statement. It's not necessary for a marker to have the same name as the host variable it represents. The only restriction is that you precede marker names with a colon (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.

Correlating substitution markers with host variables

There are two ways that StorHouse/RM correlates substitution markers with the actual host variables they represent. The method depends on whether the SQL statement containing the host variable is a SELECT or non-SELECT statement as shown in the following table:

Statement category	StorHouse/RM replaces markers with
SELECT	Host variables supplied by the USING clause of a subsequent OPEN statement
Non-SELECT	Host variables supplied by the USING clause of a subsequent EXECUTE statement.

See pages 3-6 and 5-8 for information about OPEN and EXECUTE, respectively.

Example

The following SQL statements show two ways to represent a substitution marker in a character string. The first SELECT uses a marker name (:ss_num). The second SELECT uses a question mark (?). StorHouse/RM interprets both methods the same way.

5

Using dynamic SQL

Scenarios for using dynamic SQL

```
"select name, dept_id from customer_tab where ssnum=:ss_num"
"select name, dept_id from customer_tab where ssnum=?"
```

The markers :ss_num and ? are merely place holders for a host variable that is supplied by your program in a subsequent OPEN statement.

Scenarios for using dynamic SQL

To simplify learning how and when to use dynamic SQL, we've constructed four scenarios according to the type of SQL statement you need to execute and the amount of information you know about its content.

As shown in the following table, each scenario is particularly useful for processing a specific category of dynamic SQL.

Scenario	Query?	Restrictions
1	No	<p>The SQL statement contains no markers for input host variables.</p> <p>You build the statement and process it with EXECUTE IMMEDIATE. StorHouse/RM parses the statement every time it's executed.</p> <p>Example: DROP TABLE table1</p>
2	No	<p>You must know the number of input host variables and their data types at precompile time.</p> <p>You build a dynamic SQL statement and process it with PREPARE and EXECUTE. StorHouse/RM parses the statement once but can execute it many times with different values for host variables.</p> <p>Example: DELETE FROM sysadm.syssmusers WHERE accountid = :accountid_marker</p>

Scenario	Query?	Restrictions
3	Yes	<p>You know the number of select list items, the number of markers for input host variables, and the data types of input host variables at precompile time.</p> <p>You build a dynamic query and process it with PREPARE, DECLARE, OPEN, FETCH, and CLOSE.</p> <p>Example: SELECT name, id FROM tab WHERE ssnnum = :marker1</p>
4	Yes	<p>You don't know the number of select list items, the number of markers for input host variables, and the data types of input host variables until runtime.</p> <p>You build a dynamic query and process it using an SQL descriptor area (SQLDA) which is described in the section "Scenario 4: SELECT using an SQLDA Understanding the SQLDA structure definition" on page 5-12).</p> <p>Example: SELECT col1, ... FROM mytable WHERE predicate_1, ...</p>

Each successive scenario puts fewer restrictions on your program but is more complex.

Scenario 1: Non-SELECT without markers

Scenario 1 represents the simplest category of dynamic SQL: non-SELECT statements that have no markers for input host variables. Some examples of these SQL statements are:

- REVOKE DBA FROM user1
- DROP TABLE customer_table
- DROP TABLESPACE mytablespace

Scenario 1 processes SQL statements with the static SQL statement EXECUTE IMMEDIATE. This statement parses the SQL statement presented in a host variable or a statement string and executes it.

The format for EXECUTE IMMEDIATE is:

EXEC SQL

EXECUTE IMMEDIATE { :host_variable | statement_string } ;

The following example supplies the SQL statement text to EXECUTE IMMEDIATE with a host variable:

```
EXEC SQL BEGIN DECLARE SECTION ;
char sql_str [256] ;
EXEC SQL END DECLARE SECTION ;
gets (sql_str) ;
EXEC SQL EXECUTE IMMEDIATE :sql_str ;
...
```

The following example supplies the SQL statement text to EXECUTE IMMEDIATE with a statement string. Notice that you terminate the EXECUTE IMMEDIATE statement with a semicolon because it's part of an ESQL construct. You do not terminate the character string that represents the SQL statement with a semicolon. (There is no semicolon after the word mytable.)

```
EXEC SQL EXECUTE IMMEDIATE
    'drop table mytable';
```

Note: EXECUTE IMMEDIATE works best for statements that are executed only once because it parses the statement before each execution. If you plan to execute the statement more than once, use scenario 2 instead of scenario 1.

Scenario 2: Non-SELECT with markers

Scenario 2 is for non-SELECT statements that contain markers for input host and indicator variables. You must know the data types of your input host variables at precompile time.

You use the PREPARE and EXECUTE statements to process these non-SELECTs. Both PREPARE and EXECUTE are embedded, or static, SQL statements that must be preceded by EXEC SQL and terminated with a semicolon.

- PREPARE parses an SQL statement for syntax errors and then assigns an identifier to the statement.
- EXECUTE executes the parsed SQL statement identified by the `statement_name`. It uses the current value of each input host variable named in the USING clause.

You prepare an SQL statement once but can execute it as often as necessary within the same transaction. If you commit or roll back the current transaction and want to re-execute the SQL statement, you must prepare it again.

About PREPARE

The format for PREPARE is:

```
EXEC SQL
    PREPARE statement_name FROM :string_variable ;
```

where `:string_variable` is defined as:

```
character_string | :host_variable
```

If you specify the SQL statement as a host variable, then you must have declared the host variable as a character array in a Declare Section.

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

```
EXEC SQL
    PREPARE delstmt FROM 'delete from sysadm.syssmusers where
    accountid=:mkrl' ;
```

The following SQL statement prepares an SQL statement represented as a host variable:

```
EXEC SQL
    PREPARE delstmt FROM :sql_str ;
```

About EXECUTE

The format for EXECUTE is:

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

The DESCRIPTOR clause uses an SQLDA, which is explained on page 5-12.

The USING clause applies to scenario 2. When StorHouse/RM executes an SQL statement, the input host variables in the EXECUTE statement USING clause replace the corresponding markers in the prepared SQL statement.

The following SQL statement is an example of a valid EXECUTE statement:

```
EXEC SQL
    EXECUTE delstmt USING :varname1 ;
```

Associating markers with host variables

Positioning of markers and host variables in their respective SQL statements is significant. The order of markers in the prepared SQL statement must match the order of host variables in the EXECUTE statement USING clause.

The following are valid PREPARE and EXECUTE statements. Each marker in the PREPARE statement has a corresponding host variable in the USING clause.

```
EXEC SQL
    PREPARE delstmt FROM 'delete from sysadm.syssmusers where
        accountid=:x or default_ts=:y';
EXEC SQL
    EXECUTE delstmt USING :x_var, :y_var;
```

The host variable `:x_var` is the first host variable in the USING clause list. It replaces the first marker `:x`. The host variable `:y_var` is the second variable in the USING clause list. It replaces the marker `:y`.

Example

The following example:

- Declares the host variables `sql_str` and `acct_id` as character arrays
- Sets `sql_str` equal to 'delete from sysadm.syssmusers where accountid=:mkr'
- Uses the marker `mkr` as a place holder for the host variable `acct_id`
- Parses the SQL statement represented by `sql_str`
- Assigns the statement_name `delstmt` to the prepared SQL statement
- Executes the SQL represented by `delstmt` using the current value of the host variable `acct_id`

```
...
EXEC SQL
    BEGIN DECLARE SECTION ;
    char sql_str [256] ;
    char acct_id [32] ;
EXEC SQL
    END DECLARE SECTION ;
strcpy (sql_str, "delete from sysadm.syssmusers where
accountid =:mkr");
EXEC SQL
    PREPARE delstmt FROM :sql_str ;
gets (acct_id)
EXEC SQL
    EXECUTE delstmt USING :acct_id ;
...
```

Scenario 3: Fixed-list SELECTs

Scenario 3 combines scenario 2 with the SQL statements that you need to control a cursor. Use scenario 3 for fixed-list SELECT statements when you know the following at precompile time:

- Number of items in the select list
- Number of markers for input host variables
- Data types of input host variables

You don't need to know the names of the tables and columns in your query or the values of your input host variables until runtime.

You use PREPARE plus the cursor control statements DECLARE, OPEN, FETCH, and CLOSE to process your query. See "About PREPARE" on page 5-7 for an explanation of PREPARE. See Chapter 3, "Submitting queries in ESQL" for the format descriptions of DECLARE, OPEN, FETCH, and CLOSE. The following table provides a summary description of these statements.

SQL statement	Function
DECLARE	<p>Defines a cursor and associates it with a specific query. Cursor names must be unique.</p> <p>Example: DECLARE my_cursor CURSOR FOR sel_stmt</p>
OPEN (with the USING clause)	<p>Allocates a cursor, obtains the addresses of input host variables, executes the query, and identifies the result set. The USING clause specifies the input host variables that replace substitution markers in the prepared dynamic SQL statement.</p> <p>Example: OPEN my_cursor USING :host_variable</p>
FETCH (with the INTO clause)	<p>Returns one row or an array of rows from the active set, assigns column values in the select list to corresponding host variables, and advances the cursor to the next row.</p> <p>Example: FETCH my_cursor INTO :name_var, :salary_var</p>
CLOSE	<p>Terminates cursor processing and frees system resources.</p> <p>Example: CLOSE my_cursor</p>

The following example shows how to process an SQL statement with two items in the select list and one input host variable with scenario 3.

```
extern int hereitis (
    char *tablename,
    char *acct_id )
{
    ...
EXEC SQL BEGIN DECLARE SECTION;
    char targ_acct [12];
    char dept_num [20];
    char emp_num [10];
    char sel_stmt [120];
EXEC SQL END DECLARE SECTION;
strcpy (sel_stmt, "SELECT dept_num_col,emp_num_col FROM ");
strcat (sel_stmt, table_name );
strcat (sel_stmt, " WHERE ACCT_ID = ?");
strcpy (targ_acct, acct_id);
    ...
EXEC SQL
    WHENEVER SQLERROR GO TO report_err;
EXEC SQL
    PREPARE sql_query FROM :sel_stmt;
EXEC SQL
    DECLARE my_cursor CURSOR FOR sql_query;
EXEC SQL
    OPEN my_cursor USING :targ_acct;
for (;;)
{
    EXEC SQL
        FETCH my_cursor
        INTO :dept_num, :emp_num,;
    if (sqlca.sqlcode != 0)
        break;

    /*
     *User code to process dept_num and emp_num.
     */
}
EXEC SQL
    CLOSE my_cursor ;
EXEC SQL
```

```

        COMMIT WORK;

...
    return (0);
report_err:
    printf ("Error %ld\n", sqlca.sqlcode);
    return (-1);
EXEC SQL WHENEVER SQLERROR CONTINUE;
}

```

Scenario 4: SELECT using an SQLDA

Scenario 4 is the most complicated method to code. It's also the most flexible from an application perspective. You use scenario 4 when you don't know the number of select list items, the number of markers for input host variables, and the data types of input host variables until runtime.

Before StorHouse/RM can process your dynamic SELECT, it needs specific information about each select list item and input host variable. Specifically, StorHouse/RM needs to know the:

- Number of select list items and input host variables
- Length of each
- Data type of each
- Buffer address of each

StorHouse ESQL applications use an SQL descriptor area (SQLDA) to obtain this information.

About an SQL descriptor area (SQLDA)

An *SQLDA* is a program data structure that supplies information about dynamic SQL statements. Applications create an SQLDA and allocate the correct amount of storage for SQL statement variables. The OPEN, EXECUTE, and FETCH statements use an SQLDA to obtain these storage addresses.

There are two types of SQLDAs:

- An *input SQLDA* (also called a *bind descriptor*) stores the descriptions of input host and indicator variables and the addresses of input buffers that hold the values of these variables.
- An *output SQLDA* (also called a *select descriptor*) stores descriptions of select list items and the addresses of output buffers that hold the names and values of these items.

Storing information in an SQLDA

Your application, tpe_da_xxx functions, and the DESCRIBE statement store information in an SQLDA. See “Allocating an SQLDA” on page 5-21 for information about how each field in the SQLDA is initialized.

DESCRIBE is a static SQL statement with two formats:

- DESCRIBE BIND VARIABLES
- DESCRIBE SELECT LIST

The following tables explain each format.

DESCRIBE BIND VARIABLES

Definition	Stores the number of input host variables in an SQLDA.
Format	EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name INTO input_sqlda_pointer;
Restrictions	Execute DESCRIBE BIND after you PREPARE the SQL statement represented by statement_name and before you OPEN the associated cursor. Always use the following form of OPEN with DESCRIBE BIND: EXEC SQL OPEN cursor_name USING DESCRIPTOR input_sqlda_pointer;

DESCRIBE SELECT LIST

Definition	<p>Stores the following information about output items in an SQLDA:</p> <ul style="list-style-type: none"> ■ Number of output items that are returned ■ Data type of each output item ■ Length of each output item ■ Precision value for each DECIMAL (NUMERIC) output item ■ Scale value for each DECIMAL (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 ;
Restrictions	Execute DESCRIBE SELECT LIST at any time after PREPARE.

Understanding the SQLDA structure definition

The following StorHouse SQLDA structure definition, `tpe_sqlda`, is automatically included in any program that is preprocessed by ESQL. Additionally, the structure `tpe_sqlvar` contains an entry for each variable in the SQLDA.

```

struct tpe_sqlda {
    char sqldaaid[8]; /* Eye-catcher 'TPE_SQLD' */
    uint8_t sqldvrsn; /* SQLDA version... must be 0 */
    uint8_t sqldfmod; /* Flag: fetch mode */
    int16_t sqldsize; /* Number of entries allocated */
    int16_t sqldnvar; /* Number of entries in use */
    int16_t sqldrsvl; /* (reserved) */
    int16_t sqldnrow; /* Number of rows to be fetched */
    int16_t sqldvnl; /* Maximum varname length */
    tpe_sqlvar* sqldvar; /* tpe_sqlvar elements */
};

```


Component	Description
sqldaid	Enhances the ability to validate an SQLDA and aids in debugging, making it possible to find an SQLDA in a dump. The value is TPE_SQLD.
sqldvrsn	Identifies the version of the tpe_sqlda. The version described here is 0.
sqldfmod	Contains a flag indicating the method to use for fetch operations. Values are: <ul style="list-style-type: none"> ■ 0 (TPE_DA_STANDARD) indicates a standard fetch into buffers provided by the client application. ■ 1 (TPE_DA_POINTER) indicates a pointer-fetch. Values are fetched into a buffer controlled by the server, and pointers in the SQLDA are updated to reference that buffer.
sqldsize	Contains the allocated size of the SQLDA, expressed as a number of entries. This differs from the number of active entries. The value must be ≥ 1 and \leq TPE_MAX_FIELDS.
sqldnvar	Contains the number of active entries in the SQLDA. This number must be ≥ 0 and \leq sqldsize. If a DESCRIBE operation sets this to a negative number, the absolute value of sqldnvar is the actual number of entries required for the DESCRIBE.
sqldrsv1	Is reserved for future use and must be 0.
sqldnrow	Contains the number of rows to be returned. This number must be ≥ 1 . The sqldfmod flag determines how values are returned.
sqldvnln	Contains the maximum length of a variable name when returned from StorHouse to the client. Any non-NULL sqlvname field in an entry points to an area that is at least sqldvnln bytes long.
sqldvar	Contains a pointer to the array of entries.

The structure definition for `tpe_sqlvar` is:

```
struct tpe_sqlvar {
    int32_t sqlvln32; /* Variable (maximum) length */
    int32_t* sqlvlenp; /* Pointer to actual length */
    int32_t sqlvbl32; /* Buffer length (pointer mode) */
    void* sqlvdata; /* Pointer to data */
    int16_t* sqlvind; /* Pointer to indicator variable */
    int16_t sqlvtype; /* Variable type (TPE_DT_XXX) */
    uint8_t sqlvprec; /* Precision */
    uint8_t sqlvscal; /* Scale */
    uint8_t sqlvisnl; /* Nullable flag */
    uint8_t sqlvrsv1[3]; /* (reserved) */
    char* sqlvname; /* Variable name */
    int32_t sqlvrsv2[2]; /* (reserved) */
};
```

Component	Description
<code>sqlvln32</code>	<p>Contains the maximum length for the variable. For fixed-length data, this is the actual length of the variable. For variable-length data, this includes the size of any length field:</p> <ul style="list-style-type: none"> ■ 2 bytes for VARCHAR ■ 8 bytes for BLOB and CLOB
<code>sqlvlenp</code>	<p>(Valid only for BLOB and CLOB data types)</p> <ul style="list-style-type: none"> ■ If non-zero, contains a pointer to a buffer with the 64-bit length of the LOB data. Only the low-order 32 bits are valid at this time. ■ If zero (NULL), begins with a 64-bit field, of which the low-order 32 bits are the length. ■ For a standard array fetch (<code>sqldnrow > 1</code>, <code>sqldfmod == TPE_DA_STANDARD</code>), contains a pointer to a client-supplied array of 64-bit length fields, one for each row fetched.
<code>sqlvbl32</code>	<p>(Valid only for VARCHAR, VARBINARY, BLOB, and CLOB, data types) Contains the total length of the data for the variable. StorHouse/RM sets this field during a SELECT or a FETCH operation in both standard and pointer-fetch modes. In standard array fetch mode, the value does not include any padding between array values.</p>

Component	Description
sqlvdata	Contains a pointer to the buffer with the variable data. For BLOB and CLOB data types, when sqlvlenp is NULL, the buffer begins with the 64-bit length of the data (only the low-order 32 bits are valid at this time); otherwise, sqlvdata points to the first byte of the data. For a standard array fetch (sqldnrow > 1, sqldfmod == TPE_DA_STANDARD), this must point to an application-supplied data buffer large enough to contain sqldnrow values. The length of the values is determined by sqlvln32.
sqlvind	<p>Contains a pointer to the 16-bit indicator variable.</p> <ul style="list-style-type: none"> ■ For array fetch (sqldnrow > 1), this points to an array of 16-bit indicator variables, one for each row fetched. The application must allocate indicator variables even in pointer-fetch mode. ■ For input variables, a NULL value for sqlvind is equivalent to specifying an indicator variable with a value of 0. This indicates that the accompanying value is not NULL. ■ For output variables, a NULL value for sqlvind may be used but could result in an error if one of the values retrieved is NULL or a value is truncated. A NULL value for sqlvind, for an output variable, may only be reliably used when the output variable is described as not-nullable (the DESCRIBE output for the variable indicates sqlvisnl == 0). <p>When fetching rows, StorHouse/RM sets an indicator to:</p> <ul style="list-style-type: none"> ■ 0 if the returned value is not NULL and the destination buffer is large enough to hold the value ■ > 0 if the returned value is not NULL but the buffer is not large enough for the value ■ -1 if the returned value is NULL
sqlvtype	Contains the type of the variable. The value is one of the TPE_DT_xxx values defined in sql_lib.h.
sqlvprec	Contains the precision (maximum number of digits) for numeric types.
sqlvscal	Contains the scale (number of digits to the right of the decimal point) for numeric types.
sqlvisnl	Contains a flag indicating the null-ability of the variable. If the flag is zero, then the variable is NOT NULL.

5

Using dynamic SQL

Scenarios for using dynamic SQL

Component	Description
sqlrvsv1	Is reserved for future use and must be 0.
sqlvname	Contains a pointer to an application-supplied data area at least sqldvln bytes long. If this pointer is non-NULL, the name of the host variable or column is returned as a null-terminated string. The maximum length of the string therefore is sqldvln-1.
sqlrvsv2	Is reserved for future use.

Setting values of sqlvln32 and sqlvtype fields

The following table provides the values for sqlvln32 and sqlvtype for StorHouse data types and their corresponding C data types.

C language data type	StorHouse data type	Value of sqlvln32	Value of sqlvtype
char	CHAR	Maximum number of characters in value	1
tpe_num_t	NUMERIC	24	2
short	SMALLINT	2	3
long	INTEGER	4	4
float	REAL	4	5
double	DOUBLE	8	6
tpe_date_t	DATE	sizeof (tpe_date_t)	7
tpe_time_t	TIME	sizeof (tpe_time_t)	9
tpe_timestamp_t	TIMESTAMP	sizeof (tpe_timestamp_t)	10
unsigned char	BINARY	Maximum number of bytes in value	12
unsigned char	VARBINARY	Maximum column size plus 2	17
char	VARCHAR	Maximum column size plus 2	19

C language data type	StorHouse data type	Value of sqlvln32	Value of sqlvtype
tpe_clob_t	CLOB	Maximum column size plus 8	20
tpe_blob_t	BLOB	Maximum column size plus 8	21
tpe_blob_file_t	BLOB_FILE	sizeof (tpe_blob_file_t)	22
tpe_clob_file_t	CLOB_FILE	sizeof (tpe_clob_file_t)	23
tpe_blob_loc_t	BLOB_LOCATOR	sizeof (tpe_blob_loc_t)	24
tpe_clbo_loc_t	CLOB_LOCATOR	sizeof (tpe_clob_loc_t)	25

Resetting, or coercing, data types

You can reset, or coerce, the data type of a particular output variable. You coerce the data type of a specific variable by resetting the corresponding `sqlvtype` field in the `sqldvar` array. For example, to coerce the data type of your variable [3], reset `sqldvar[3].sqlvtype` to the desired data type.

Do not coerce BINARY, VARBINARY, and VARCHAR data types to CHAR. Coercion from VARBINARY or VARCHAR to CHAR invalidates the 2-byte length field that precedes variable length data. Coercion from BINARY to CHAR causes an unsupported data conversion.

Note: If you coerce a data type, the StorHouse extractor is not used. See Chapter 7, “Using the StorHouse extractor,” for more information about the StorHouse extractor.

Checking space for SQLDA entries

The `tpe_da_getnbytes` function calculates the size, in bytes, of an SQLDA that may contain size entries. The definition of `tpe_da_getnbytes` is:

```
tpe_status_t tpe_da_getnbytes (int16_t size, int32_t*
nbytes)
```

The following example uses `tpe_da_getnbytes` with `tpe_da_setup`. See “Initializing storage as an SQLDA” on page 5-22 for more information about `tpe_da_setup`.

```
int16_t nvars = 10;
int32_t nbytes;
tpe_sqlda* newda;
tpe_da_getnbytes (nvars, &nbytes);
newda = (tpe_sqlda*)malloc (nbytes);
tpe_da_setup (newda, nvars, TPE_DA_APPEND_ENTRIES);
```

Checking space for variable name data

The `tpe_da_getvnbytes` function calculates the size, in bytes, of the space required to store variable name (`sqlvname`) data. Use this routine when you wish to use your own allocation function. The input `tpe_sqlda` must be initialized with the correct number of variables (`sqldsize`). The `size` argument contains the length (including any null terminator) of the variable name entries.

The definition of `tpe_da_getvnbytes` is:

```
tpe_status_t tpe_da_getvnbytes (tpe_sqlda* da, int16_t size,
int32_t* nbytes)
```

Here is an example using `tpe_da_getvnbytes`. See also “Allocating variable entries in an SQLDA” on page 5-22.

```
int32_t    nvars = 10;
int16_t    vnsz = 32; // each varname is 32 bytes long
int32_t    nbytes;
int32_t    i;
tpe_sqlda* newda;
char*      vndata;

tpe_da_getnbytes (nvars, &nbytes);
newda = (tpe_sqlda*)malloc (nbytes);
tpe_da_setup (newda, nvars, TPE_DA_APPEND_ENTRIES);
tpe_da_getvnbytes (newda, vnsz, &nbytes);
vndata = (char*)malloc (nbytes);
```

```
for (i = 0; i < nvars; i++)
{
    newda->sqldvar[i].sqlvname = vndata;
    vndata += vnsiz;
}
```

Allocating an SQLDA

The `tpe_da_alloc` function allocates an SQLDA structure. Specifically, this function allocates, initializes, and returns a pointer to an SQLDA that may contain entries up to the provided size.

The definition of `tpe_da_alloc` is:

```
tpe_status_t tpe_da_alloc (int16_t size, tpe_sqlda** da)
```

The `tpe_da_alloc` function initializes the SQLDA components as follows:

SQLDA component	Set to
sqldaid	TPE_SQLD
sqldsize	The value of the 'size' argument
sqldnvar	The value of the 'size' argument
sqldnrow	1
sqldfmod	0
sqldvrsn	0
sqldrsv1	0
sqldvnln	0
sqldvar	Point to an allocated array of entries
array of entries	0

The following sample code allocates an SQLDA that may contain up to 20 entries:

```
...
tpe_sqlda *sqldaptr;
tpe_status_t rc;
...
rc = tpe_da_alloc (20, &sqldaptr);
if (rc != STATUS_OK)
{
    fprintf (stderr, "Error allocating SQLDA: %d\n", rc);
    exit (0);
}
```

Allocating variable entries in an SQLDA

The `tpe_da_alloc_varnames` function allocates space for variable (column) names and sets the appropriate fields in the `tpe_sqlda`. The `sqldsize` and `sqldvar` values in the `tpe_sqlda` must be set correctly prior to using this function.

The definition of `tpe_da_alloc_varnames` is:

```
tpe_status_t tpe_da_alloc_varnames (tpe_sqlda* da, int16_t
size)
```

The following sample code allocates up to 30 variables:

```
rc = tpe_da_alloc_varnames (sqldaptr, 30);
if (rc != STATUS_OK)
{
    fprintf (stderr, "Error allocating SQLDA: %d\n", rc);
    exit (0);
}
```

Initializing storage as an SQLDA

The `tpe_da_setup` function initializes storage as an SQLDA. You would use `tpe_da_setup` (instead of `tpe_da_alloc`) in conjunction with `tpe_da_getnbytes` and

your own allocation routine (for instance, malloc). No action is taken if the SQLDA pointer is NULL, or if size ≤ 0. If entries is NULL (or TPE_DA_APPEND_ENTRIES), tpe_da_setup assumes that the data area in da is sufficiently large to contain the variable entries. The tpe_da_getnbytes function returns the correct length for such an area.

The definition of tpe_da_setup is:

```
tpe_status_t tpe_da_setup (tpe_sqlda* da, int16_t size,
tpe_sqlvar* entries)
```

The tpe_da_setentry function initializes the SQLDA components as follows:

SQLDA component	Set to
sqldaaid	TPE_SQLD
sqldsiz	The value of the 'size' argument
sqlnvar	The value of the 'size' argument
sqlnrow	1
sqldfmod	0
sqldvrsn	0
sqldrsv1	0
sqldvnln	0
sqldvar	The value of the 'entries'

Setting values in an SQLDA variable entry

The tpe_da_setentry function sets values in an SQLDA variable entry. Specifically, this function sets the data type, length, indicator variable pointer and data pointer to the values provided. All other fields in the entry are left untouched.

The definition of `tpe_da_setentry` is:

```
tpe_status_t tpe_da_setentry (const tpe_sqlda* da, int16_t
entry,
int16_t type, int32_t len, int32_t* indptr, char* dataptr)
```

Freeing an SQLDA

You can free (or delete) a previously allocated SQLDA by calling the `tpe_da_free` function. Note that this function frees space for variable names only if you allocated them with the `tpe_da_alloc_varnames` function.

The definition of `tpe_da_free` is:

```
tpe_status_t tpe_da_free (tpe_sqlda* da)
```

Note: Always free allocations for data buffers and indicator variable buffers before calling `tpe_da_free`. Passing an invalid argument to `tpe_da_free` or freeing an already freed SQLDA could cause memory corruption.

Checking the size of your SQLDA

Always check that the SQLDA you allocated is large enough. Initially, `sqlnvar` contains the value of the size argument that you specified in `tpe_da_alloc`. This number represents your estimate of the number of input host variables for `DESCRIBE BIND` or the number of output items for `DESCRIBE SELECT` in your dynamic SQL statement.

The `DESCRIBE` statement determines the actual number of input host variables or output items in your SQL statement and stores that number in the SQLDA field `sqlnvar`. A positive value in `sqlnvar` indicates that the SQLDA is large enough to hold information for all your variables. A negative value indicates that the SQLDA is too small for your variables. The absolute value of this negative number represents the actual number of host variables found by `DESCRIBE`. (For example, if `sqlnvar` contains -11, the actual number of host variables found by `DESCRIBE` is 11.) If `sqlnvar` is negative after you execute `DESCRIBE`, then you

must free the SQLDA, reallocate it large enough for your input host variables or output items, and then re-DESCRIBE the SQL statement.

The following example:

- Allocates an input SQLDA with a size of 20
- Issues DESCRIBE BIND to set the actual number of input variable references
- Checks sqldnvar for negative
- Reallocates the SQLDA with sqldsize equal to sqldnvar

```
static int  static_stmt( const char *p_user, const char *p_tsName )
{
    int  ix;
    int  nvars = -1;
    int  daSize = 20;
    struct tpe_sqlda *isqlda;
    const char *sqlFunc = "";

    EXEC SQL BEGIN DECLARE SECTION;
    char stmtStr[ 100 ];
    char p1[ 13 ];
    char p2[ 33 ];
    EXEC SQL END DECLARE SECTION;

    // Set up Error condition Handler

    EXEC SQL WHENEVER SQLERROR GOTO feterr;

    strcpy( stmtStr,
            "insert into sysadm.syssmusers (accountid, default_ts)"
            "values (:p1, :p2)" );
    strcpy( p1, p_user );
    strcpy( p2, p_tsName );

    sqlFunc = "PREPARE";
    EXEC SQL
        PREPARE insStmt FROM :stmtStr;

    while ( nvars < 0 )
    {
        if ( tpe_da_alloc( daSize, &isqlda ) != STATUS_OK )

        {
            printf( "No memory 1\n" );
```

5**Using dynamic SQL**Scenarios for using dynamic SQL

```

        return ( -1 );
    }

    sqlFunc = "DESCRIBE";
    EXEC SQL
        DESCRIBE BIND VARIABLES FOR  insStmt INTO isqlda;

    nvars = isqlda->sqldnvar;
    if ( nvars < 0 )
    {
        daSize = -nvars;
        tpe_da_free( isqlda );
    }
}

for ( ix = 0 ; ix < nvars ; ++ix )
{
    isqlda->sqldvar[ix].sqlvtype = TPE_DT_CHAR;
    isqlda->sqldvar[ix].sqlvind  = (short *)0;
    isqlda->sqldvar[ix].sqlvprec = 0;
    isqlda->sqldvar[ix].sqlvscal = 0;
    isqlda->sqldvar[ix].sqlvisnl = 0;
}

isqlda->sqldvar[0].sqlvln32 = strlen( p1 );
isqlda->sqldvar[0].sqlvdata = (void *)p1;
isqlda->sqldvar[1].sqlvln32 = strlen( p2 );
isqlda->sqldvar[1].sqlvdata = (void *)p2;

sqlFunc = "EXECUTE";
EXEC SQL
    EXECUTE insStmt USING DESCRIPTOR isqlda;

sqlFunc = "COMMIT";
EXEC SQL
    COMMIT WORK;

printf( "Static ins statement prepared successfully \n" );

EXEC SQL WHENEVER SQLERROR CONTINUE;

return ( 0 );

```

```
feterr:
    fprintf( stderr, "SQL Error in %s sqlcode=%ld, %s\n",

            sqlFunc, sqlca.sqlcode, sqlca.sqlerrm );

    EXEC SQL
        ROLLBACK WORK;

    return ( -1 );
}
```

Allocating SQLDA buffers for data and indicator variables

The `tpe_da_alloc` function does not allocate buffers for host and indicator variables. Your application must allocate these buffers separately and set pointers to them in the SQLDA. For input SQLDAs, always set the SQLDA components `sqlvtype` and `sqlvln32` before you allocate buffers and allocate buffers before you execute OPEN. For output SQLDAs, `DESCRIBESELECTLIST` sets `sqlvln32` and `sqlvtype` for you. Just be sure to allocate output buffers before you execute FETCH.

Two functions are provided to help you allocate and manage data and indicator variable buffers:

- `tpe_da_getbsize`
- `tpe_da_setptrs`

Calculating the buffer size (`tpe_da_getbsize`). The `tpe_da_getbsize` function calculates and returns (in `blen`) the length of the buffer required to hold the data described in the provided `tpe_sqlda`. This length includes space for indicator variables.

The definition of `tpe_da_getbsize` is:

```
tpe_status_t tpe_da_getbsize (const tpe_sqlda* da, int32_t*
blen)
```

An example follows the `tpe_da_setptrs` section.

Initializing buffer pointers (tpe_da_setptrs). The tpe_da_setptrs function initializes the buffer pointers (sqlvdata, sqlvind) in the provided SQLDA. This function uses the information in the SQLDA to divide the buffer into smaller buffers. Each of these smaller buffers is large enough to hold the data for one variable described in the SQLDA. The sqlvlenp and sqlvind pointers for each variable in the SQLDA are set to point to a piece of the buffer. It is the caller's responsibility to ensure that buff is sufficiently large (see tpe_da_getbsize).

For example, suppose the SQLDA describes three variables:

- Integer (TPE_DT_INTEGER)
- VARCHAR(14) (TPE_DT_VARCHAR, sqlvln32==16)
- Double-word floating-point value (TPE_DT_DOUBLE)

In the following example, the buffer offsets and buffer length are provided for explanation purposes only. No application should be coded so that it depends on this behavior. The tpe_da_setptrs function sets:

- sqldvar[0]->sqlvdata to the address of the buffer
- sqldvar[1]->sqlvdata to the address of the buffer+4
- sqldvar[2]->sqlvdata to the address of the buffer+20

It then sets the pointers to the indicator variables:

- sqldvar[0]->sqlvind to the address of buffer+28
- sqldvar[1]->sqlvind to buffer+30
- sqldvar[2]->sqlvind to buffer+32

The buffer is divided into 6 buffers of lengths 4, 16, 8, 2, 2, 2. So for a single-row fetch, the buffer required is just over 34 bytes.

The definition of tpe_da_setptrs is:

```
tpe_status_t tpe_da_setptrs (tpe_sqlda* da, void* buff)
```

The following sample code allocates buffers for data and indicator variables.

```

...
/*
 * Allocate buffer space for data and indicators.
 * sqldaptr is an SQLDA that has been initialized by
 * a DESCRIBE or similar function.
 */
int32_t buflen;
void* buffer;

tpe_da_getbsize (sqldaptr, &buflen);
buffer = malloc (buflen);
if (buffer == NULL)
{
    fprintf (stderr, "No memory!\n");
    ...
}
tpe_da_setptrs (sqldaptr, buffer);

```

Using multiple SQLDAs

If you have multiple cursors open at the same time for dynamic queries, you need a separate SQLDA for each cursor operation. Unlike a cursor or a connection, an SQLDA is not a "named" object known to the precompiler. You simply create as many SQLDA structures as you need, assigning the pointer returned for each to a separate variable. The FETCH will use whichever SQLDA is pointed to in the USING DESCRIPTOR clause.

Reusing the same SQLDA

Non-concurrent cursors can reuse the same SQLDA as long as the maximum sqldsize is sufficient for all queries.

Understanding the status value

SQLDA functions return status values (tpe_status_t) that are identical to the sqlcode values in an SQLCA. These status codes are documented in the *StorHouse SQL Reference Manual*. Generally, all of the routines return 0 (STATUS_OK) on a

successful completion. The routines that allocate memory (such as `tpe_da_alloc` and `tpe_da_alloc_varnames`,) may return -20001 to indicate insufficient memory.

Reviewing the basics

Let's review the basics necessary to process a dynamic SELECT statement with scenario 4. We'll make the following assumptions.

- Your SELECT statement contains one substitution marker for an input host variable in the WHERE clause and multiple column identifiers in the select list.
- Your program prompts the user for the SELECT statement text and for values of all input host variables at runtime.
- The SELECT statement text will be similar to the following statement string:

```
select ..., ... from emptable where deptnum = :marker_num
```

- Because you won't know information about your host variables until runtime, you'll need to use input and output SQLDAs to supply your program with the data types, lengths, and storage addresses of these variables.

Here are the SQL statements that you'll use to process this type of dynamic query:

- EXEC SQL
PREPARE statement_name FROM :string_variable;
- EXEC SQL
DECLARE cursor_name CURSOR FOR statement_name;
- EXEC SQL
DESCRIBE BIND VARIABLES FOR statement_name
INTO input_sqlda_pointer;

- EXEC SQL
OPEN cursor_name USING input_sqllda_pointer;
- EXEC SQL
DESCRIBE SELECT LIST FOR statement_name
INTO output_sqllda_pointer;
- EXEC SQL
FETCH cursor_name USING DESCRIPTOR output_sqllda_pointer;
- EXEC SQL
CLOSE cursor_name;
- EXEC SQL
COMMIT WORK;

Here's a list of the ESQL-related steps you need to include in your program:

- Declare input and output host variables in a Declare Section.
- Allocate and set up an input and output SQLDA.
- PREPARE the query from the host variable that stores the statement string.
- DECLARE a cursor for the query.
- DESCRIBE the input host variable into the input SQLDA.
- Allocate storage for the input host variable found by DESCRIBE.
- OPEN the cursor using the input SQLDA.
- DESCRIBE the select list into the output SQLDA.
- Allocate storage for select list items and indicator variables.
- FETCH a row into the allocated data buffers using the output SQLDA.
- CLOSE the cursor.
- Commit the transaction to release all locks and resources.
- Deallocate the storage that was used for variables.
- Deallocate the SQLDAs.

Satisfying individual program requirements

The program that satisfies the previous assumptions uses input and output SQLDAs. Your program may have different requirements. It may not need all the

SQL statements listed on page 5-30 to process queries, or it may need to combine one statement format from scenario 3 with another statement format from scenario 4. Dynamic SQL gives you the flexibility to mix and match.

Using an SQLDA for array fetches

StorHouse/RM supports two methods for using dynamic SQL to fetch multiple rows at a time into host arrays: standard and pointer-fetch.

Note: Do not use the BLOB_FILE and CLOB_FILE data types with any multiple row fetch method. If these types are present, the `sqldnrow` field of the SQLDA must be 1; otherwise, an error occurs.

About the standard method

The standard method for fetching multiple rows requires that you set pointers to your host and indicator variables and that you allocate memory for the data to be fetched. You must allocate data buffers and indicator variable buffers that are large enough to hold the maximum number of rows of maximum size in your result set. Then you set the SQLDA component `sqldnrow` to the maximum number of rows to be retrieved per FETCH call. If you set `sqldnrow` to 50 and your result set contains 60 rows, the first FETCH retrieves 50 rows. The second FETCH retrieves 10 rows. Each FETCH moves result set values to the data buffers that you allocated.

The following example allocates memory and sets pointers for `sqlvdata` and `sqlvind` for array fetches:

```
/* set the array size in sqlda */
sqldaptr->sqldnrow = 10;
...
/*
 * Allocate buffer space for data and indicators.
 * sqldaptr is an SQLDA that has been initialized by
```

```

    * a DESCRIBE or similar function.
    */

int32_t  buflen;
void*    buffer;

tpe_da_getbsize (sqldaptr, &buflen);
buffer = malloc (buflen);
if (buffer == NULL)
{
    fprintf (stderr, "No memory!\n");
    ...
}
tpe_da_setptrs (sqldaptr, buffer);

```

About the pointer-fetch method

The pointer-fetch method is a StorHouse/RM performance enhancement. With this method, FETCH sets pointers to result set data in its current location rather than copying values to your buffer. Therefore, you don't need to allocate an output buffer for host variables, and you don't need to set sqlvdata because FETCH sets it for you. If you use indicator variables, you must still allocate a buffer for these values and point sqlvind to that buffer.

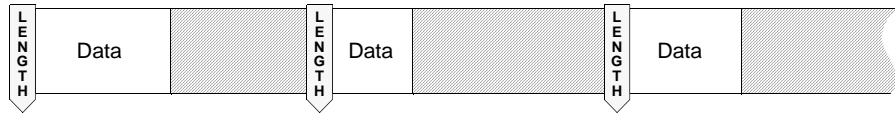
To use the pointer-fetch method, set the SQLDA field sqldfmod to 1 (TPE_DA_POINTER). You must also set sqldnrow to the maximum number of rows you want to retrieve with each FETCH.

Consider the following when using the pointer-fetch method:

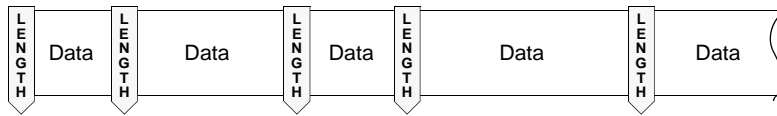
- When you FETCH result set values, you must process these values before you fetch your next set of rows. Each subsequent FETCH may overlay the previous set of rows with new data.
- A FETCH *may* return rows and sqlcode 100 in the same fetch, for instance, when the cursor is already pointing to the last row of the active set. See page

3-10 for an example. An extractor cursor operation, however, returns the partial buffer with sqlcode 0 and an empty buffer with sqlcode 100.

- The standard and pointer-fetch methods use different strategies for looping through VARCHAR and VARBINARY data.
 - With the standard method, you use the maximum field length plus 2 bytes as the offset to the next data field.



- Data is contiguous in the buffer with the pointer-fetch method. To loop through your data, simply read the 2-byte length that precedes each data field and use its value (actual length) plus 2 bytes as the offset to the next data field.



Sample program

The following sample program shows how to use the pointer-fetch method.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int32_t fetch_example( const char *p_stmtstring /* Ptr for SELECT
*/ );

static int  user_process( long p_row,
                        int  p_col,
                        void *,          /* Current data item ptr */
                        int32_t,         /* Length of that item */
                        int16_t          /* and its type */

```

```

    );
static int32_t usage( const char *prog );

static int32_t usage( const char *p_prog )
{
    fprintf( stderr, "Usage: %s <sysid> <dbname>\n\n", p_prog );
    return 0;
}

int main( int argc, char *argv[] )
{
    const char *funcid;          /* Current function name for err */
    int rc = 0;                  /* Return code, 0 (ok) or -1 */

    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[80];
    EXEC SQL END DECLARE SECTION;

    if ( argc != 3 )
        return ( usage( argv[0] ) );

    /*
     * Copy the database name to a variable known to the
     * precompiler.
     */
    strcpy( dbname, "filetek:T:" );
    strcat( dbname, argv[1] );
    strcat( dbname, ":" );
    strcat( dbname, argv[2] );

    /*
     * Set up Error Condition handler
     */
    EXEC SQL WHENEVER SQLERROR GOTO err;

    /*
     * Connect to the specified database
     * with the connection name conn1
     */
    funcid = "CONNECT";
    EXEC SQL CONNECT TO :dbname AS 'conn1';

    /*
     * Call function to execute a SELECT statement

```

```

    */
    rc = fetch_example( "SELECT * FROM sysadm.sysindexes" );

    /*
     * Disconnect from the database
     */
    funcid = "DISCONNECT";
    EXEC SQL DISCONNECT 'conn1';
    return rc;

EXEC SQL WHENEVER SQLERROR CONTINUE;

err:
    printf( "SQL Error %d in function %s\n Text: %s\n",
           sqlca.sqlcode, funcid, sqlca.sqlerrm );
    return -1;
}

static int32_t fetch_example( const char *p_stmtstring )
{
    long rowix;           /* Row index with a one FETCH */
    long rownum = 1;      /* Row number (1 for first fetched) */
    long rowcurr;         /* Current row number */
    int colix;            /* Indexes to current row/column */
    short datalen;        /* Data length of curr data value */
    int16_t datatype;     /* Type of data for a column */
    void *dataptr;        /* Pointer to current data value */
    tpe_sqlda *sqldaptr; /* Pointer to SQLDA structure */
    const char *funcid;   /* Current function name for err */

EXEC SQL BEGIN DECLARE SECTION;
char stmt[500];          /* For input SELECT statement */
EXEC SQL END DECLARE SECTION;

    /*
     * Set up an SQLDA for 64 return columns. For this
     * example we assume this is always enough.
     */
    tpe_da_alloc( 64, &sqldaptr );

    /*
     * Set up the (local) Error Condition handler
     */
EXEC SQL WHENEVER SQLERROR GOTO err;

```

```
/*
 * Put the input SQL statement (which must be SELECT)
 * into a variable known to the ESQL precompiler.
 */
strcpy( stmt, p_stmtstring );

/*
 * Prepare the dynamic SQL statement, declare a cursor to
 * process the returned data, and then open that cursor.
 */
funcid = "PREPARE";
EXEC SQL PREPARE query_stmt FROM :stmt;

funcid = "DECLARE";
EXEC SQL DECLARE the_cursor CURSOR FOR query_stmt;

funcid = "OPEN";
EXEC SQL OPEN the_cursor;

/*
 * Describe the output variables. This sets up the type
 * and length arrays in the SQLDA.
 */
funcid = "DESCRIBE";
EXEC SQL DESCRIBE SELECT LIST FOR query_stmt INTO sqldaptr;

/*
 * Set up the SQLDA to use pointer-fetch mode, and to
 * fetch up to 100 rows per operation.
 */
sqldaptr->sqldfmod = TPE_DA_POINTER;
sqldaptr->sqldnrow = 100;

/*
 * We don't need to set up variable pointers, but we
 * still need to set up indicator variable pointers. In
 * this case we assume the data is NOT NULL and zero out
 * these pointers.
 */
for ( colix = 0 ; colix < sqldaptr->sqldnvar ; ++colix )
    sqldaptr->sqldvar[colix].sqlvind = NULL;

/*
 * Fetch the data. The outer loop is for the entire
 * FETCH cycle. Each fetch returns an array of rows.
 * The inner loops process each column (variable) and
 * within that, each individual data item.
 */
```

```

    */
    funcid = "FETCH";
    while ( sqlca.sqlcode == 0 )
    {
        EXEC SQL FETCH the_cursor USING DESCRIPTOR sqldaptr;
        for ( colix = 0 ; colix < sqldaptr->sqldnvar ; ++colix )
        {
            dataptr = sqldaptr->sqldvar[colix].sqlvdata;
            datatype = sqldaptr->sqldvar[colix].sqlvtype;
            datalen = sqldaptr->sqldvar[colix].sqlvln32;

            for ( rowix = 0 ; rowix < sqlca.sqlerrd[3] ; ++rowix )
            {
                rowcurr = rownum + rowix;
                dataptr = (void*)((char *)dataptr +
                                user_process( rowcurr, (colix + 1),
                                                dataptr, datalen, datatype
));
            }

            rownum += sqlca.sqlerrd[3];
        }

        funcid = "CLOSE";
        EXEC SQL CLOSE the_cursor;

        funcid = "COMMIT";
        EXEC SQL COMMIT WORK;

        tpe_da_free( sqldaptr );
        return 0;

    err:
        printf ("SQL Error %d in function %s\n Text: %s\n",
                sqlca.sqlcode, funcid, sqlca.sqlerrm );
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK WORK;
        tpe_da_free( sqldaptr );
        return -1;
    }

static int  user_process (
                long p_row,
                int  p_col,
                void *p_dataptr, /* Current data item ptr */

```



```
        int32_t p_datalen, /* Length of that item */
        int16_t p_datatype /* and its type */
    )
{
    int  actlen = p_datalen;
    short varlen = p_datalen;
    char *dptr = (char *)p_dataptr;

    if ( (p_datatype == TPE_DT_VARCHAR) ||
        (p_datatype == TPE_DT_VARBINARY) )
    {
        memcpy( (void *)&varlen, p_dataptr, 2 );
        actlen = varlen + 2;
        dptr += 2;
    }

    printf( "Data: Row: %ld Col: %d Len: %d Type: %d ",
            p_row, p_col, (int)varlen, (int)p_datatype );

    if ( (p_datatype == TPE_DT_VARCHAR) ||
        (p_datatype == TPE_DT_CHAR) )
        printf( "Value: <%.s>\n", varlen, dptr );
    else
        printf( "Firstbyte: 0X%02X\n", *(unsigned char *)dptr );

    return ( actlen );
}
```

5

Using dynamic SQL

Using an SQLDA for array fetches

Accessing large objects

This chapter describes basic information about accessing large objects (LOBs) and explains how to:

- Place LOB data into a host variable
- Use a locator variable to access and manipulate LOB data
- Place LOB data into a client file

Ways to access LOB values

A *LOB value* is the content of a LOB column in a row in a user table. A LOB value may be a *binary large object* (BLOB) or a *character large object* (CLOB) up to 2 GB in size.

You can refer to and manipulate LOB values using host variables just as you would any other data type. For instance, you can define a host variable as a BLOB or CLOB data type when your program needs the entire LOB value. Host variables, however, use the client memory buffer which may not always be large enough to hold a LOB value. StorHouse ESQL supports two additional ways to access LOB values:

- Locator variables
- File reference variables

Locator variables

A *locator variable* is a type of host variable that refers to a LOB value or LOB expression on StorHouse. You define locator variables with the BLOB_LOCATOR and CLOB_LOCATOR data types. By using a locator variable, ESQL programs can manipulate the LOB value—at the server—as if the value was stored in a regular host variable. The value associated with the locator variable is valid until the end of the transaction or until you explicitly free it with the FREE LOCATOR statement.

Locator variables are useful when:

- A program needs only a part of a LOB value.
- The entire LOB value cannot fit in the program's memory or in a regular host variable.
- A program needs a temporary LOB value from a LOB expression but does not need to save the result.

File reference variables

A *file reference variable* is a type of host variable useful for transferring a LOB value to a client file. You define file reference variables with the BLOB_FILE and CLOB_FILE data types. The file referenced by the file reference variable must be accessible from the system on which the program runs. When using file reference variables, you must set file options in the file reference variable structure. See the BLOB_FILE specification on page 2-7 or the CLOB_FILE specification on page 2-10 for more information about file options.

StorHouse/RM currently does not support input file reference variables to move data from a client file to StorHouse. Also, you cannot use a BLOB_FILE or CLOB_FILE file reference variable for the following:

- A multi-row fetch operation (array or pointer)
- A pointer-fetch operation, even if only one row is fetched

Sample LOB value

The following LOB value is a product description in a product catalog table. Each product description is an XML document defined as a CLOB. The examples in this chapter access and manipulate this LOB value.

```
<Product>
<Name>Turkey Wrench</Name>
<Developer>Gobble Labs, Inc.</Developer>
<Summary>Like a monkey wrench, but smaller.</Summary>
<Description>
<Para>The turkey wrench, which comes in both right- and left-
handed versions (skyhook optional), is made of the finest
stainless steel. The Read-i-grip rubberized handle quickly
adapts to your hands, even in the greasiest situations. </
Para>
<Para>You can:</Para>
<List>
<Item><Link URL="Order.html">Order your turkey wrench</
Link></Item>
<Item><Link URL="Wrenches.htm">Read more about wrenches</
Link></Item>
<Item><Link URL="catalog.zip">Download the catalog</Link></
Item>
</List>
<Para>The turkey wrench costs just $19.99 and, if you order
now, comes with a hand-crafted shrimp hammer as a bonus
gift.</Para>
</Description>
</Product>
```

The CREATE TABLE definition for the product catalog is as follows:

```
CREATE TABLE ProdInfo
  (ProdNo INTEGER,
   ProdName VARCHAR(100),
   ProdDescr CLOB(2 MB) );
```

Placing LOB data into a host variable

You can place LOB data into a host variable that is large enough to hold the data.

To do this, you:

- Define a host variable as a BLOB or CLOB data type
- Issue a query to access the data and place the result into the host variable

For example, assume you want to access all product descriptions in the ProdInfo table. First, define a host variable to contain the result:

```
EXEC SQL BEGIN DECLARE SECTION;  
      CLOB hv_product_desc;  
EXEC SQL END DECLARE SECTION;
```

Then issue the query, placing the result into the host variable:

```
SELECT ProdDescr  
      INTO :hv_product_desc  
      FROM ProdInfo;
```

Using a locator variable to select LOB data

When querying a user table containing LOB data, you can use a locator variable to manipulate a LOB value at the server and to fetch that value or part of the value to the client as needed. To do this, you:

- Declare the locator variable
- Issue the query and associate a LOB value with the locator variable
- Manipulate the LOB value, if needed, through the locator variable
- Release the locator variable

For example, suppose you need to search the ProdInfo table to find a wrench that comes with a bonus gift.

Declaring a locator variable

In order to use a locator variable, you must first declare one with the BLOB_LOCATOR or CLOB_LOCATOR data type in a Declare Section. If you intend to retrieve a LOB value or part of it back to the client, you must also define a host variable for the value with the BLOB or CLOB data type. See “Defining StorHouse data types” on page 2-6 for more information about these data types. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
    long hv_start_descr;
    long hv_end_descr;
    long hv_bonus_start;
    CLOB_LOCATOR hv_prod_locator;
    CLOB_LOCATOR hv_prod_desc_locator;
    CLOB(2M)hv_product;
EXEC SQL END DECLARE SECTION;
```

The preceding Declare Section defines the following:

- hv_start_descr and hv_end_desc host variables to contain the starting and ending position of the product description.
- hv_bonus_start host variable to contain the starting position of the bonus information.
- hv_prod_locator locator variable to be associated with a specific product in the user table.
- hv_prod_desc_locator locator variable to be associated with the product description of the product.
- hv_product host variable to contain the resulting CLOB value on the client.

You can continue to use a locator variable as long as you have not released it with the FREE LOCATOR statement and the transaction has not ended.

Issuing the query

When using a locator variable, you issue the query and associate the result with a locator variable. The LOB value remains on StorHouse. The associated locator value moves to the client.

- For a static query that selects one row, use the SELECT statement INTO clause to identify the locator variable.
- For a static query that selects multiple rows, use a cursor and FETCH statement INTO clause to identify the locator variable.

Example using SELECT INTO

The following example, using SELECT INTO, associates the result with the locator variable :hv_prod_locator.

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

Example using FETCH

The following example, using a cursor and FETCH statement, declares and opens a cursor called my_cursor and associates the result with the locator variable :hv_prod_locator.

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

EXEC SQL
      OPEN my_cursor;
```



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

Manipulating a LOB value through a locator variable

For a static query, you can manipulate or evaluate a LOB value that's associated with a locator variable by using the VALUES INTO statement.

The format for VALUES INTO is:

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

For example, the following VALUES INTO statements use the INSTR function to locate the starting and ending position of the product description associated with the locator variable :hv_prod_locator. The starting position is placed into host variable :hv_start_descr and the ending position is placed into host variable :hv_end_descr.

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

Then these VALUES INTO statements use the SUBSTR and INSTR functions to locate the substring bonus gift.

```
EXEC SQL VALUES
    ( SUBSTR(:hv_prod_locator,
        :hv_start_descr,
        :hv_end_descr - :hv_start_descr)
    INTO :hv_prod_desc_locator;
EXEC SQL VALUES
    ( INSTR(:hv_prod_desc_locator, "bonus gift")
    INTO :hv_bonus_start;
```

Finally, this VALUES INTO statement places the resulting product description into the host variable :hv_product. At this point, the LOB value moves to the client.

```
if ( hv_bonus_start != npos ) {  
    EXEC SQL VALUES (:hv_prod_desc_locator)  
    INTO :hv_product;  
}
```

Releasing a locator variable

You can release a locator variable before the end of a transaction by using the FREE LOCATOR statement. This statement removes the association between one or more locator variables and their values and frees the storage used by a locator variable at the StorHouse server. If you don't explicitly release locator variables, StorHouse/RM releases them at the end of the transaction.

The format for FREE LOCATOR is:

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

For example, the following statement releases two locator variables.

```
EXEC SQL  
    FREE LOCATOR :hv_prod_locator , :hv_prod_desc_locator ;
```

Placing LOB data into a client file

You can use a file reference variable to place a LOB value into a client file. For instance, an ESQL program can access an audio clip stored in a BLOB column, write the clip into the file, and start the audio player with the file as input. You can create a file, overwrite data in a file, or append data to a file.

To place LOB data from StorHouse into a client file, you:

- Declare the file reference variable
- Initialize the client file variable
- Issue the query

Assume you are querying the same product information table used in the previous section. The query places a specific product description into a client file named ProdFile.xml.

Declaring a file reference variable

In order to place data into a client file, you must first declare a file reference variable with the BLOB_FILE or CLOB_FILE data type in a Declare Section. See “Defining StorHouse data types” on page 2-6 for more information about these data types.

For instance, the following example defines a file reference variable named ProdFile:

```
EXEC SQL BEGIN DECLARE SECTION;  
      CLOB_FILE ProdFile;  
EXEC SQL END DECLARE SECTION;
```

Note that you cannot use a BLOB_FILE or CLOB_FILE file reference variable for the following:

- A multi-row FETCH (array or pointer)
- A pointer FETCH, even if only one row is fetched

Initializing the client file variable

Then before you can retrieve the data from StorHouse, your program must

initialize the following fields for the client file variable:

- `file_options` – Must be one of the following:
 - `TPE_FILE_CREATE` – Creates a new file. An error occurs if the file exists.
 - `TPE_FILE_OVERWRITE` – Replaces any existing file.
 - `TPE_FILE_APPEND` – Appends fetched data to an existing file or creates a new file.
- `name` – The name of the file.
- `name_length` – The length of the file name (in bytes). The maximum length of a file name is 255.

For example:

```
ProdFile.file_options = TPE_FILE_CREATE;  
strcpy (ProdFile.name , "/home/user/ProdFile.xml");  
ProdFile.name_length = strlen(ProdFile.name);
```

Note: The application sets the `data_length` portion of the file reference variable to the length of the new data written to the file.

Issuing the query

Finally, issue the query and associate the result with the file reference variable. If the query selects one row, use the `SELECT` statement `INTO` clause to identify the file reference variable. If the query selects multiple rows, use a cursor and a `FETCH` statement to identify the file reference variable.

Example using `SELECT INTO`

The following example, using `SELECT` and the `INTO` clause, associates the result (the XML product description for product number 123) with the file reference variable `:ProdFile`.

```
EXEC SQL
SELECT ProdDescr
      INTO :ProdFile
      FROM ProdInfo
      WHERE ProdNo = 123;
```

Example using FETCH

The following example, using a cursor and FETCH statement INTO clause, associates the result with the file reference variable :ProdFile.

```
EXEC SQL
DECLARE my_cursor CURSOR FOR
    SELECT ProdDescr
    FROM ProdInfo
    WHERE ProdNo = 123;

EXEC SQL OPEN my_cursor;

EXEC SQL
    FETCH my_cursor
    INTO :ProdFile;
```

In the previous examples, StorHouse/RM places the data directly into the /home/user/PDFile.xml file. The application might, for example, then start an XML reader with the file name as a parameter. The length of the document is in PDFile.data_length. If more than one row is fetched, and you want to store each CLOB value, you must change the file name (ProdInfo.name) before each FETCH.

Using the StorHouse extractor

This chapter describes the StorHouse extractor software: what it does, the types of queries that qualify for extractor processing, and the extractor requirements. It also explains what to check in the SQLCA to determine whether a database engine or the extractor processed a query.

About the StorHouse extractor

The *StorHouse extractor* provides fast path processing for some queries that result in full table scans and full segment selects.

- A *full table scan* reads every row in a table, without the use of an index, to determine a result set.
- A *full segment select* returns data from one or more entire segments with the use of a range index.

The extractor processes full table scans and full segment selects quicker and more efficiently than the standard StorHouse engine. Furthermore, when a table resides on both StorHouse optical and tape media, the extractor always uses the tape copy when it's available to benefit from faster sequential I/O.

Types of eligible queries

Two types of queries are eligible for extractor processing: simple and full segment. You can submit these queries as embedded or dynamic SQL. StorHouse/RM decides whether to process queries with the extractor at PREPARE time.

Simple queries

A *simple query* is a SELECT statement that requires a full table scan. Simple queries must meet the following requirements as well as the extractor requirements on page 7-4.

- The query cannot contain a WHERE, ORDER BY, GROUP BY, or DISTINCT clause.
- The StorHouse account issuing the query must have the SCAN database privilege to perform a full table scan.

Format

```
SELECT simple_column_list FROM table
```

where simple_column_list is defined as:

```
* | column_name [,column_name]...
```

Example

```
SELECT col1, col2 FROM tabl
```

The extractor processes this query if all requirements are met.

Full segment queries

A *full segment query* is a SELECT statement that returns one or more entire table segments with the use of a range index. If a user table consists of multiple segments and a query selects both complete segments and partial segments, the extractor processes the complete segments and a database engine processes the partial segments. Full segment queries must meet the following requirements as well as the extractor requirements on page 7-4.

- All predicates in the WHERE condition must be based on columns in range indexes.
- All predicates must select all rows from one or more table segments.

Format

```
SELECT simple_column_list FROM table  
WHERE condition
```

where simple_column_list is defined as:

```
* | column_name [,column_name]...
```

and condition is one or more predicates.

Example

```
SELECT * FROM janbilling  
WHERE billdate BETWEEN '01/15/2000' AND '03/15/2000'
```

The extractor may process this query if the billdate column is indexed in a range index, the predicate selects at least one complete table segment, and all other extractor requirements are met.

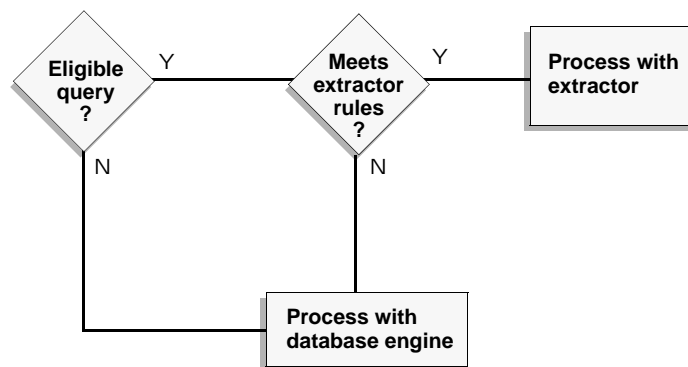
Qualifying for extractor processing

Eligible queries must also meet the following requirements to qualify for extractor processing.

- The query must not contain any subqueries.
- The query must refer to one user table.
- The host and StorHouse systems must have the same native values key so that no byte-reordering of INTEGER and SMALLINT columns is necessary.
- The application issuing the query must not have changed any values in SQLDA fields set by DESCRIBE.

Note: The extracted columns of the table can be NULL or NOT NULL and any database data type. Some client machines, however, may not support DECIMAL, REAL, and DOUBLE data types.

If an eligible query doesn't satisfy these requirements, StorHouse/RM processes it with a database engine instead of the extractor. The following drawing summarizes how StorHouse/RM qualifies eligible queries for extractor processing.



For instance, StorHouse/RM selects the database engine rather than the extractor to process the following query because the query contains more than one table reference:

```
SELECT * FROM tab1, tab2
```

Checking the SQLCA

The SQLCA field `sqlerrd [5]` indicates whether StorHouse/RM used the extractor or a database engine to process a query. StorHouse/RM records this information in the SQLCA at PREPARE time.

When this method is used	StorHouse/RM puts this value in <code>sqlerrd [5]</code>
Extractor	Non-zero
Database engine	0

Do not test for a specific value to determine whether the extractor is used, that is, check for only 0 or non-zero.

7

Using the StorHouse extractor

Checking the SQLCA

Managing transactions

This chapter describes transaction management in ESQL. It defines atomic and durable operations, general transaction guidelines, and locking. It explains when transactions start and how to end them with COMMIT WORK or ROLLBACK WORK.

About transactions

A *transaction* is one or more SQL statements that are treated as a single unit of work. A *unit of work* is a recoverable sequence of operations. In StorHouse, transactions are both atomic and durable. *Atomic* means that all operations in the transaction are either *committed* (applied to the database) or *rolled back* (canceled, or not applied to the database). *Durable* means that once a transaction is committed, the changes done by that transaction are permanent.

The following transaction guidelines apply:

- If an SQL statement fails, StorHouse/RM removes all effects of that statement from the database.
- An application must decide whether to commit or roll back a transaction. A commit saves all SQL statements that executed successfully within the transaction. In contrast, a rollback cancels the entire transaction.
- When a failed SQL statement is part of a larger transaction, prior SQL statements in the transaction are not affected.

- Typically an application rolls back an entire transaction whenever one of the SQL statements in the transaction fails.

Starting a transaction

A transaction begins with the first SQL statement that affects the state of the database or the transaction itself. A PREPARE statement, for example, affects the state of a transaction because it creates a prepared statement that's used only for the duration of that transaction. A CREATE TABLE statement affects the state of a database because it updates the metadata. Executable statements (see the table on page 1-4 for a list) affect the state of a database, so they can begin a transaction. In contrast, the following SQL statements do not affect the state of a database or transaction, therefore, they do not begin a transaction:

- CONNECT
- BEGIN DECLARE SECTION
- END DECLARE SECTION
- WHENEVER

When an active transaction already exists, an executable statement executes as part of that active transaction. If there is no active transaction, then the statement starts a new transaction, which then becomes the active transaction. All subsequent SQL statements execute as part of the active transaction until the application ends the transaction by explicitly committing it or rolling it back.

For example, assume an application just CONNECTed to submit a dynamic query. There's no current active transaction. A dynamic query consists of several SQL statements, including PREPARE, DECLARE CURSOR, OPEN, and so on. PREPARE is the first SQL statement that affects the state of the transaction, so the active transaction begins with PREPARE and the remaining SQL statements become part of the active transaction until it's committed or rolled back.

For example:

```
CONNECT
No active transaction
  PREPARE (begin active transaction)
  DECLARE CURSOR
  OPEN
  DESCRIBE SELECT LIST
  FETCH
  CLOSE
  COMMIT WORK (end active transaction)
No active transaction
```

Ending a transaction

A transaction ends when it is committed or rolled back, either explicitly with a COMMIT WORK or ROLLBACK WORK statement or implicitly for DDL statements.

Committing a transaction

The COMMIT WORK statement terminates a transaction when all SQL statements complete successfully. COMMIT WORK:

- Releases all locks held by a transaction
- Closes all open cursors within a transaction
- Frees any active locator variables
- Makes any changes to the database permanent

An example of a COMMIT WORK statement is:

```
EXEC SQL COMMIT WORK ;
```

Committing DDL statements

StorHouse/RM implicitly issues a COMMIT WORK statement before and after every DDL statement (ALTER, CREATE, DROP, GRANT, REVOKE). An application cannot roll back changes to a StorHouse database after issuing DDL statements.

Committing non-DDL statements

Non-DDL statements like PREPARE and SELECT also place locks on tables (see page 8-5 for locking information), so it's necessary to release those locks by issuing a COMMIT WORK statement. An application must explicitly issue a COMMIT WORK statement for non-DDL statements to end those transactions. An application can roll back the effect of non-DDL statements before committing a transaction, but once committed, the transaction cannot be canceled.

Rolling back a transaction

The ROLLBACK WORK statement—typically used in exception handlers—cancels any changes made to the database within an uncommitted transaction.

ROLLBACK WORK:

- Releases all locks held by a transaction
- Closes all open cursors within a transaction
- Frees any active LOB locators

An example of a ROLLBACK WORK statement is:

```
EXEC SQL ROLLBACK WORK ;
```


Automatic rollback

A rollback occurs automatically when:

- An application terminates abnormally, for instance, in the event of a hardware or software failure or a lock time-out error
- An active transaction exists when an ESQL application disconnects from a database

When a serious error occurs, StorHouse/RM might implicitly mark a transaction for rollback. The SQLCA identifies transactions implicitly marked for rollback (see “sqlwarn” on page 4-3). In this case, the application must roll back the current transaction before proceeding with execution of the next SQL statement.

Locking

StorHouse supports the *serializable* ANSI/ISO transaction isolation level, which guarantees the highest read consistency and data integrity in a database. StorHouse implements table-level locks to support this isolation level. Tables include user tables, system tables, and views.

A commit or rollback is *always* required to release table-level locks, even for queries. This section describes the types of locks, the operations that require locks, and the duration of locks.

Types of locks

There are two types of table-level locks: shared and exclusive.

Shared locks

A *shared* (or read) *lock* reserves a table for reading only. This lock prevents a table from being dropped. Multiple engines can have a shared lock on the same table.

A metadata backup uses a special database-level lock that has the effect of read-locking every system table. The following operations require shared locks on user tables:

- Loads (at the beginning and end (no locks during data transfer))
- DDL processing
- Queries

Exclusive locks

An *exclusive* (or write) *lock* reserves a table for updating only. This lock prohibits a table from being shared. One engine can have an exclusive lock on a table. All other lock requests (shared and exclusive) for the table are queued.

DML operations require an exclusive lock on system table views, and the following operations require an exclusive lock on system tables:

- Metadata recovery
- Loads (at start and end)
- DDL processing

The following SQL statements require an exclusive lock on user tables:

- CREATE TABLE
- CREATE INDEX
- DROP TABLE
- DROP INDEX

Duration of a lock

An engine holds on to locks—both shared and exclusive—against user tables throughout a transaction. It releases shared locks against system tables as soon as it's done processing the system tables, and it releases exclusive locks against system tables when the transaction ends.

8

Managing transactions

Locking

Using the ESQL precompiler

The ESQL precompiler translates each static SQL statement into C code to build a program that can be compiled. After you precompile your source file, you run the standard C compiler to create an object file from the C source file. Then you link the object files with the ESQL libraries to create the application executable.

This chapter explains what's required to precompile, compile, and link ESQL programs. These topics include:

- Setting default values for StorHouse environment variables
- Issuing the `esqlc` command with the proper command options

Setting environment variables

StorHouse/RM uses the following environment variables to supply default values for the StorHouse/RM root directory and various precompiler options.

Environment variable	Indicates
DB_NAME	(optional to override the current default) The name of your default database.
DB_PASSWD	(optional to override the current default) The password of your default StorHouse account.
DB_USER	(optional to override the current default) Your default StorHouse account ID.

Environment variable	Indicates
ESQL_CC	(optional, depending on which operating system you use) The command string required to invoke the desired C compiler (cc, acc, CC, etc.). You must include the -c option (compile only).
ESQL_CPP	(optional, depending on which operating system you use) The C preprocessor command and basic options.
ESQL_LINK	(optional, depending on which operating system you use) The command string that is required to invoke the linker. ESQL programs must use the C++ linker.
STHROOT	(required) The directory where the StorHouse ESQL product is installed. This directory must contain the StorHouse ESQL bin and lib subdirectories.

The default compile (ESQL_CC) and link (ESQL_LINK) commands differ by operating system. The Sun Solaris default compile and link commands are:

Solaris default compile and link commands

Variable	Default command
ESQL_CC	cc -c -I<sth_dir>/include/ <app_program.c>
ESQL_LINK	CC -o <app_program> <app_program.o> /<sth_dir>/lib/libsthfe.a -lm -lnsl -lsocket

Note that the compile is run using `cc`, the Sun ANSI C compiler. The link is done with `CC`, the Sun Workshop C++ compiler. The compiler is used here just to set up and run the linker. To override the default and use C++ as the compiler for Solaris, set ESQL_CC as follows:

```
setenv ESQL_CC "CC -c -g"
```

The HP default compile and link commands are:

HP default compile and link commands

Variable	Default command
ESQL_CC	cc -Ae -c -I<sth_dir>/include/ <app_program>.c
ESQL_LINK	aCC -AA -o <app_program> <app_program>.o /<sth_dir>/lib/libsthfe.a -lm -lnsl

To override the compile default and use C++ as the compiler for HP, set ESQL_CC as follows:

```
setenv ESQL_CC "aCC -c -AA"
```

Issuing the esqlc command

You invoke the ESQL precompiler with the `esqlc` command. The ESQL precompiler passes all command line options that it does not recognize to the C compiler. This feature allows you to specify C compiler options directly as options to the `esqlc` command. Options that are recognized by the ESQL precompiler are prefixed by the plus symbol (+) to avoid clashes with options supported by the C compiler and linker.

If you don't use any of the + options listed in the following table, then the `esqlc` command precompiles and compiles the specified source file with the same line numbering as your .pc file. In addition, you can't use `#define` symbols in your ESQL statements.

The format of the esqlc command is:

esqlc [option_list] file_name_list

Argument	Description
option_list	(optional) Specifies different ways to run the command.
+G	Inserts debugging code in the generated .c files and adds the -g option to the c compiler.
+L	Suppresses redefinition of source line numbers in the generated C code.
+K	Keeps all intermediate files generated during compilation.
+T	Translates the embedded SQL source code (.pc files) to C source code (.c files). There is no additional processing of the generated C source code.
+P	Runs the C preprocessor on the input file before translating the SQL statements in the input file so that #define'd symbols can be used in ESQL statements.
+V	Displays the commands invoked by the precompiler as it processes files.
-c	Generates .o files from .pc or .c files without linking.
-o	Generates an executable named 'filename' from .pc and .c files.
file_name_list	<p>Specifies the name(s) of the files to include in the precompile operation. Valid file extensions are:</p> <ul style="list-style-type: none"> ■ .pc - indicates embedded SQL source code files. ■ .c - indicates C source code files. ■ .o - indicates object files.

Some examples follow.

- To build the application executable orders from the orders.pc source file, execute the following esqlc commands:

```
esqlc -c orders.pc  
esqlc -o orders orders.o
```

In this example, the ESQL precompiler precompiles the source file and then invokes the C compiler to compile the generated .c file. The second command invokes the system linker to link the object file into an executable named orders.

- If SQL statements in the orders.pc source file use #define'd symbols, then execute the following esqlc commands:

```
esqlc +P orders.pc  
esqlc -o orders orders.o
```

9

Using the ESQL precompiler

Issuing the esqlc command

Reviewing a sample program

This appendix contains one ESQL sample program that illustrates how to perform static SELECT statements.

```
/*
 * Example program showing usage of static select statements.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int static_select( const char * );
static int usage( const char *prog );

static int usage( const char *prog )
{
    fprintf( stderr, "Usage: %s <sysname> <dbname>\n\n", prog );
    return ( 0 );
}

main
(
    int argc,
    char *argv[]
)
{
    int rc = 0;

    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[64];
    EXEC SQL END DECLARE SECTION;
```

```
if ( argc != 3 )

    return ( usage( argv[0] ) );

/*
 * Copy the database name
 */
strcpy( dbname, "filetek:T:" );
strcat( dbname, argv[1] );
strcat( dbname, ":" );
strcat( dbname, argv[2] );

/*
 * Set Error condition handler
 */
EXEC SQL
    WHENEVER SQLERROR GOTO err;

/*
 * Connect to the specified database
 * with the connection name conn1
 */
EXEC SQL
    CONNECT TO :dbname AS 'conn1';

/*
 * Call function to execute a static select statement
 */
rc = static_select( "SYS" );

/*
 * Disconnect from the database
 */
EXEC SQL
    DISCONNECT 'conn1';

return ( rc );

EXEC SQL
    WHENEVER SQLERROR CONTINUE;

err:
printf( "SQL Error (%ld) %s\n", sqlca.sqlcode, sqlca.sqlerrm );

return ( -1 );
```

```
}

/*
 * static_select : demonstrates the usage of
 * a static select statement. Gets the list of
 * tables whose names do not have a prefix 'SYS'.
 */
static int static_select
(
    const char *p_pat
)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tretval[33];
    char likepat[9];
    EXEC SQL END DECLARE SECTION;

    /*
     * Set up Error condition Handler
     */
    EXEC SQL
        WHENEVER SQLERROR GOTO err;

    EXEC SQL
        DECLARE stcur CURSOR FOR
        SELECT tbl
        FROM sysadm.systables
        WHERE tbl NOT LIKE :likepat;

    /*
     * Note: For static statements, if the declare cursor
     * statement contains references to auto variables,
     * OPEN statement should be in the same C function.
     */
    EXEC SQL
        WHENEVER NOT FOUND GOTO over;

    strcpy( likepat, "%%%%%%%%" );
    strncpy( likepat, p_pat, strlen( p_pat ) );

    EXEC SQL
        OPEN stcur USING :likepat;

    for ( ; ; )
```

A**Reviewing a sample program**

```
{
    tretval[0] = '\0';
    EXEC SQL
        FETCH stcur INTO :tretval;
    printf( "<%s>\n", tretval );
}

EXEC SQL
    WHENEVER NOT FOUND CONTINUE;

over:
    EXEC SQL
        CLOSE stcur;
    EXEC SQL
        COMMIT WORK;

    printf( "Static select statement executed successfully\n" );

    return ( 0 );

    EXEC SQL
        WHENEVER SQLERROR CONTINUE;

err:
    fprintf( stderr, "SQL Error: %d %s\n", sqlca.sqlcode,
        sqlca.sqlerrm );

    EXEC SQL
        ROLLBACK WORK;

    return ( -1 );
}
```

Converting and comparing data

This appendix explains two StorHouse/RM API functions:

- `tpe_conv_data` to explicitly convert data from one type to another
- `tpe_compare_data` to compare two values with the same database data type

Converting data

StorHouse/RM performs the required data conversions between data types wherever possible. While delivering each SELECT list value to host variables in the INTO clause, StorHouse/RM implicitly converts the data value, if necessary, from the database representation to the host language representation. For example, if you retrieve a database column of type SMALLINT into a host variable of type long, StorHouse/RM automatically converts the value from SMALLINT to long integer.

StorHouse/RM does not allow implicit type conversions between all data types. For example, if an integer host variable holds a character value, then StorHouse/RM does not perform an implicit data conversion and reports an error.

The StorHouse/RM API call `tpe_conv_data` explicitly converts data from one type to another. The syntax of this call is:

```
tpe_status_t tpe_conv_data (  
    int16_t,          /* IN  input type          */  
    int32_t,          /* IN  length of input buffer */
```

B**Converting and comparing data**

Converting data

```

    const void*,      /* IN  input buffer          */
    const char*,      /* IN  conversion format    */
    int16_t,          /* IN  output type desired  */
    int32_t,          /* IN  length of output buf */
    void*             /* OUT output buf for result */
);

```

The data types supported by StorHouse/RM API calls are:

- TPE_DT_CHAR
- TPE_DT_DATE
- TPE_DT_DOUBLE
- TPE_DT_INTEGER
- TPE_DT_NUMERIC
- TPE_DT_REAL
- TPE_DT_SMALLINT
- TPE_DT_TIME
- TPE_DT_TIMESTAMP
- TPE_DT_VARCHAR

The following sample code shows the conversion from character data to integer data:

```

EXEC SQL BEGIN DECLARE SECTION ;
    char data [20] ;
    long outval ;
EXEC SQL END DECLARE SECTION ;

strcpy (data, "1234") ;
status = tpe_conv_data (TPE_DT_CHAR, strlen (data), data,
    "", TPE_DT_INTEGER, sizeof (long), &outval) ;

```

In the preceding example:

- data is the buffer containing input data
- outval is the output data that is returned

Note: For better portability, SQL statements can be used for data type conversion in ESQL. On the other hand, using the StorHouse/RM API call `tpe_conv_data` improves the performance of the ESQL program.

Comparing data

You can compare two data values with the same database data type with the StorHouse/RM API call `tpe_compare_data`. The syntax of this call is:

```
tpe_status_t tpe_compare_data (
    int16_t,          /* IN  data type          */
    int32_t,          /* IN  length of LHS value */
    const void*,      /* IN  LHS value          */
    int32_t,          /* IN  length of RHS value */
    const void*,      /* IN  RHS value          */
    int16_t*          /* OUT comparison result   */
);
```

The result is a short integer with the following values:

This result value	Indicates that the
1	First data value is greater than the second
-1	First data value is less than the second
0	Two data values are equal

The data types supported for comparison are:

- TPE_DT_CHAR
- TPE_DT_DATE
- TPE_DT_INTEGER
- TPE_DT_NUMERIC
- TPE_DT_REAL
- TPE_DT_SMALLINT

- TPE_DT_TIME
- TPE_DT_TIMESTAMP
- TPE_DT_VARCHAR

The function `tpe_compare_data` returns a zero (0) if the comparison is successful; otherwise, it returns a non-zero error code.

The following sample code shows the comparison of two numeric data elements:

```
EXEC SQL BEGIN DECLARE SECTION ;
    numeric val1 ;
    numeric val2 ;
    short result ;
EXEC SQL END DECLARE SECTION ;

...
status = tpe_compare_data (TPE_DT_NUMERIC, sizeof
(tpe_num_t),
    (void *)&val1, sizeof (tpe_num_t), (void *)&val2,
    &result) ;
if (status == 0)
{
    if (result > 0)
        printf ("val1 > val2 ") ;
    else
        if (result < 0)
            printf ("val1 < val2 ") ;
        else
            printf ("val1 = val2 ") ;
}
```

The previous example compares `val1` and `val2` and returns the result in `result`.

Note: For better portability, SQL statements can be used for comparison of data in ESQL. On the other hand, using the StorHouse/RM API call `tpe_compare_data` improves the performance of the ESQL program.

Developing ESQL applications

This appendix lists some tips for writing ESQL programs. More tips will be added in subsequent revisions of this manual.

Coding guidelines

Here are some guidelines and tips for writing ESQL applications:

- Statements represented as character strings must be null-terminated. However, no nulls may occur within the string.
- The `WHENEVER` statement provides more flexibility and reduces the code size of an application when it's used in an ESQL program to check for error conditions.
- Don't use the `+P` option on the `esqlc` command if any SQL reserved words are `#define'd` in some of the included header files. When the `+P` option is used in `esqlc`, the source files are passed through the C preprocessor before the ESQL statements are translated.

One frequent problem is the reserved word `NULL`, which is `#define'd` in the standard header file `stdio.h`. In this case, you can:

- Avoid using the `+P` option.
- Avoid the inclusion of the header file that contains the definition of the reserved word.

- Reverse the case of the reserved word when used in the ESQL statement, as shown below. This does not affect esqlc because reserved words in ESQL statements are not case sensitive.

```
EXEC SQL
    SELECT ename
    FROM employee
    WHERE commission IS null ;
```

- You can use #define symbols in SQL statements wherever you can use constants. The following example shows the usage of the #define symbol, MAX_SALARY:

```
#define MAX_SALARY 10000

EXEC SQL
    SELECT ename, salary
    FROM employee
    WHERE salary = MAX_SALARY ;
```

- It is good practice to specify all columns in the SELECT list instead of using an asterisk (*). The full SELECT list improves the readability of the SELECT statement. In addition, if the database table schema is ever changed, you won't have to subsequently modify the SELECT statement.
- When you issue queries, use an indicator variable to check for NULL values on columns that can contain NULL values.
- The StorHouse SQL DECODE function is similar to the switch statement in C.

Index

Symbols

() in SQL syntax xiv

+G option for esqlc command 9-4

+K option for esqlc command 9-4

+L option for esqlc command 9-4

+P option for esqlc command 9-4

+T option for esqlc command 9-4

+U option for esqlc command 9-4

+V option for esqlc command 9-4

... in SQL syntax xiv

.c file name for esqlc command 9-4

.o file name for esqlc command 9-4

.pc file name for esqlc command 9-4

{ } in SQL syntax xiv

| in SQL syntax xiv

' in SQL syntax xiv

A

account_id argument 2-30

action_sp argument for WHENEVER statement 4-7

active set, definition 1-7

ALL argument 2-32

allocating

SQLDA buffers 5-27

SQLDA structure 5-21

variable entries in an SQLDA 5-22

API functions

tpe_compare_data B-3

tpe_conv_data B-1

array

declaring variables 2-21

definition 2-21

fetches 3-9, 5-32

associating a cursor with a query 3-5

atomic, definition 8-1

automating condition checking and error handling
with WHENEVER 4-6

B

BINARY data type 2-6

BLOB data type 2-6

BLOB_FILE data type 2-7

BLOB_LOCATOR data type 2-8

braces in SQL syntax xiv

brackets in SQL syntax xiv

C

- C format for declaring a host array 2-23
- C language data types, mapping to StorHouse data types 2-4
- c option for esqlc command 9-4
- C structures
 - new_type_name 2-21
 - tpe_blob_file_t 2-8
 - tpe_clob_file_t 2-10
 - tpe_num_t 2-14
 - tpe_sqlda 5-14
 - tpe_sqlvar 5-16
 - tpe_time_t 2-16
 - tpe_timestamp_t 2-18
- C++-style comments 1-9
- calculating buffer size 5-27
- CHARACTER data type 2-9
- character strings, rules for dynamic statements 5-2
- checking the size of an SQLDA 5-24
- client file, initializing 6-9
- CLOB data type 2-9
- CLOB_FILE data type 2-10
- CLOB_LOCATOR data type 2-11
- CLOSE statement
 - description 3-9
 - example 5-10
 - format 3-9
 - to control cursors 3-5
- closed cursor, definition 3-5
- closing a cursor 3-9
- coding guidelines for writing ESQL applications C-1
- comments
 - C++-style 1-9
 - C-style 1-9
 - SQL-style 1-8
- COMMIT WORK statement
 - description 8-3
 - format 8-3, 8-4
- comparing data with the tpe_compare_data function B-3
- compiling ESQL programs 9-1
- components
 - SQLCA
 - sqlcabc 4-2
 - sqlcaid 4-2
 - sqlcode 1-8, 3-8, 4-2
 - sqlerrd 4-3, 7-5
 - sqlerrm 4-2
 - sqlerrml 4-2
 - sqlerrp 4-2
 - sqlwarn 4-3
 - SQLDA
 - sqlvln32 5-18
 - sqlvtype 5-18
- CONNECT statement
 - account_id argument 2-30
 - connection_name argument 2-29
 - database_name argument 2-29
 - description 2-29
 - example 2-30
 - format 2-29
 - password argument 2-30
- connection string for a StorHouse remote database 2-30
- connection_name argument

- CONNECT statement 2-29
 - DISCONNECT statement 2-32
 - SET CONNECTION statement 2-31
 - continuations in C 1-10
 - continuing SQL statements on another line 1-10
 - Control Center, description x
 - conventions, SQL xiv
 - converting data with the tpe_conv_data function B-1
 - correlating substitution markers with host variables 5-3
 - C-style comments 1-9
 - CURRENT argument 2-32
 - current connection, definition 2-28
 - current row, definition 1-7
 - cursor
 - associating with a query 3-5
 - closing 3-9
 - definition 1-7
 - opening 3-6
 - retrieving rows 3-7
 - rules 3-5
 - using 3-4
 - using with host variable arrays 3-9
 - cursor states
 - closed 3-5
 - open 3-5
- ## D
- data definition (DDL) statements 1-4
 - data manipulation (DML) statements 1-4
 - data types, general
 - mapping StorHouse types to C types 2-4
 - supported
 - by StorHouse/RM API calls B-2
 - for comparison B-3
 - data types, specific
 - BINARY 2-6
 - BLOB 2-6
 - BLOB_FILE 2-7
 - BLOB_LOCATOR 2-8
 - CHARACTER 2-9
 - CLOB 2-9
 - CLOB_FILE 2-10
 - CLOB_LOCATOR 2-11
 - DATE 2-11
 - DECIMAL 2-13
 - DOUBLE 2-12
 - int16_t 2-4
 - int32_t 2-4
 - int64_t 2-4
 - int8_t 2-4
 - INTEGER 2-13
 - NUMERIC 2-13
 - REAL 2-15
 - SMALLINT 2-16
 - TIME 2-16
 - TIMESTAMP 2-17
 - uint16_t 2-4
 - uint32_t 2-4
 - uint64_t 2-4
 - uint8_t 2-4
 - VARBINARY 2-19
 - VARCHAR 2-19
 - database data type, definition 1-7
 - database_name argument 2-29
 - DATE data type 2-11
 - DB_NAME environment variable 9-1

Index

D

- DB_PASSWD environment variable 9-1
- DB_USER environment variable 9-1
- DECIMAL data type 2-13
- declarative statement, definition 1-3
- Declare Section
 - declaring host and indicator variables 2-2
 - definition 1-5, 1-6, 2-1
 - example 2-2
 - rules for coding 2-2
- DECLARE statement
 - example 5-10
 - format 3-6
 - used to control cursors 3-4
- declaring
 - a host array
 - C format 2-23
 - ESQL format 2-22
 - an array as a new data type 2-24
 - host and indicator variables in a Declare Section 2-2
 - host variables as
 - C data types 2-4
 - StorHouse types 2-4
 - locator variable 6-5
 - output file reference variable 6-9
 - variables as
 - host arrays 2-21
 - type definitions 2-20
- DECODE function C-2
- definitions
 - active set 1-7
 - array 2-21
 - atomic 8-1
 - closed cursor 3-5
 - current connection 2-28
 - current row 1-7
 - cursor 1-7
 - database data type 1-7
 - declarative statement 1-3
 - Declare Section 1-5, 1-6, 2-1
 - durable 8-1
 - dynamic SQL 5-1
 - embedded SQL 1-3
 - ESQL construct 1-3
 - executable statement 1-4
 - full segment query 7-3
 - full table scan 7-1
 - host language 1-3
 - host language data type 1-7
 - host language variable 1-5
 - host program 1-3
 - indicator variable 1-6
 - input host variable 1-5
 - input SQLDA 5-13
 - open cursor 3-5
 - output host variable 1-5
 - output SQLDA 5-13
 - place holder 5-3
 - query 3-1
 - result set 1-7
 - select list 3-1
 - simple query 7-2
 - SQLCA 1-7
 - SQLDA 1-4
 - static SQL 1-3
 - StorHouse extractor 7-1
 - substitution markers 5-3
 - transaction 8-1
- delimiters 1-10
- DESCRIBE statement 5-13
- DISCONNECT statement
 - ALL argument 2-32
 - connection_name argument 2-32

- CURRENT argument 2-32
- description 2-31
- example 2-32
- format 2-31

- DOUBLE data type 2-12

- durable, definition 8-1

- dynamic SQL
 - definition 5-1
 - when to use 1-4

E

- ellipsis points in SQL syntax xiv

- embedded SQL, definition 1-3

- environment variables for StorHouse
 - DB_NAME 9-1
 - DB_PASSWD 9-1
 - DB_USER 9-1
 - ESQL_CC 9-2
 - ESQL_CPP 9-2
 - ESQL_LINK 9-2
 - STHROOT 9-2

- ESQL
 - construct, definition 1-3
 - format for declaring a host array 2-22
 - guidelines for writing applications C-1
 - precompiler 1-1
 - program path 1-2
 - sample program A-1

- ESQL_CC environment variable 9-2

- ESQL_CPP environment variable 9-2

- ESQL_LINK environment variable 9-2

- esqlc command

- examples 9-4
- file_name_list argument 9-4
- format 9-4
- issuing 9-3
- option_list argument 9-4

examples

- associating a cursor with a query 3-6
- associating markers with host variables 5-9
- BLOB data type 2-6
- BLOB_FILE data type 2-7
- BLOB_LOCATOR data type 2-8
- changing the current connection 2-31
- CHARACTER data type 2-9
- checking for status codes 4-4
- checking for warnings 4-5
- checking the size of the SQLDA 5-25
- CLOB data type 2-9
- CLOB_FILE data type 2-10
- CLOB_LOCATOR data type 2-11
- CLOSE statement 5-10
- closing a cursor 3-9
- COMMIT WORK statement 8-3
- CONNECT statement 2-30
- connecting to a StorHouse database 2-30
- correlating substitution markers with host variables 5-3
- C-style comments 1-9
- DATE data type 2-11
- DECLARE statement 5-10
- declaring an array as a new data type 2-25
- delimiters 1-10
- DOUBLE data type 2-12
- ESQL format for declaring a host array 2-22
- esqlc command 9-4
- EXECUTE statement 5-8
- FETCH statement 5-10
- FREE LOCATOR statement 6-8
- INTEGER data type 2-13
- non-SELECT statements without markers 5-5

Index

F

- NUMERIC 2-13
 - OPEN statement 5-10
 - opening a cursor 3-7
 - pointer fetch method for array fetches 5-34
 - preparing an SQL statement from a character string 5-7
 - queries that return a single row 3-2
 - REAL data type 2-15
 - retrieving multiple rows using a cursor 3-8
 - ROLLBACK WORK statement 8-4
 - SET CONNECTION 2-31
 - SMALLINT data type 2-16
 - SQL-style comments 1-9
 - standard method for array fetches 5-32
 - statement labels 1-11
 - storing a dynamic SQL statement as a host variable 5-2
 - terminating a connection 2-32
 - TIME data type 2-16
 - TIMESTAMP data type 2-17
 - using a cursor with host variable arrays 3-10
 - using file reference variables 6-8
 - using host variables 2-26, 6-3
 - using indicator variables 2-28
 - using locator variables 6-4
 - VALUES INTO statement 6-7
 - WHENEVER statement 4-8
 - exception_sp argument for WHENEVER statement 4-7
 - exclusive lock 8-6
 - EXEC SQL delimiter 1-10
 - executable statement, definition 1-4
 - EXECUTE IMMEDIATE statement
 - example 5-6
 - format 5-6
 - EXECUTE statement
 - example 5-8
 - format 5-8
 - extractor 7-1
- ## F
- FETCH statement
 - description 3-7
 - example 5-10
 - format 3-7
 - multiple rows 3-9
 - pointer fetch method 5-33
 - to control cursors 3-5
 - file reference variable
 - declaring 6-9
 - definition 6-2
 - example 6-8
 - file options 2-8
 - file_name_list file extensions for esqlc command
 - .c 9-4
 - .o 9-4
 - .pc 9-4
 - FLOAT data type 2-12
 - FREE LOCATOR statement 6-8
 - freeing an SQLDA 5-24
 - full segment query
 - definition 7-3
 - example 7-3
 - format 7-3
 - processing 7-4
 - full segment select 7-1
 - functions
 - DECODE C-2

NVL 1-5
tpe_compare_data B-1
tpe_conv_data B-1
tpe_da_free function 5-24

G

GROUP BY clause in SELECT statement 3-2

guidelines
 embedding SQL in C 1-8
 writing ESQL applications C-1

H

HAVING clause in SELECT statement 3-2

host language datatype, definition 1-7

host language variable, definition 1-5

host language, definition 1-3

host program, definition 1-3

host variable in an ESQL statement
 format 2-26
 names 1-10
 rules 2-26

I

in SQL syntax xiv

indicator variable in an ESQL statement
 definition 1-6
 format 2-27
 rules 2-27

initializing
 buffer pointers 5-28
 client files 6-9

input host variable, definition 1-5

input SQLDA, definition 5-13

int16_t data type 2-4

int32_t data type 2-4

int64_t data type 2-4

int8_t data type 2-4

INTEGER data type 2-13

INTO clause in SELECT statement 3-2

issuing the esqlc command for the ESQL precompiler
 9-3

L

large objects (LOBs)
 definition 6-1
 manipulating with VALUES INTO 6-7
 placing into a host variable 6-3
 placing LOB data into a client file 6-8
 using a locator variable 6-4
 using file reference variables 6-2

linking ESQL programs 9-1

locator variable
 definition 6-2
 manipulating 6-7
 releasing 6-8
 using to select LOB data 6-4

lock types
 exclusive 8-6
 shared 8-6

locking 8-5

lowercase in SQL syntax xiv

M

mapping StorHouse data types to C types 2-4

maximum number of open database connections 2-28

methods for fetching multiple rows
 pointer fetch 5-33
 standard 5-32

multiple SQLDAs 5-29

N

new_type_name C structure 2-21

NULL values 1-5

NUMERIC data type 2-13

NVL function 1-5

O

-o option for esqlc command 9-4

open cursor, definition 3-5

OPEN statement
 description 3-6
 example 5-10
 format 3-6
 to control cursors 3-4

opening a cursor 3-6

operators 1-11

option_list argument for esqlc command
 +G 9-4
 +K 9-4
 +L 9-4
 +P 9-4
 +V 9-4
 -c 9-4
 -o 9-4

ORDER BY clause in SELECT statement 3-2

output host variable, definition 1-5

output SQLDA, definition 5-13

P

password argument 2-30

place holder, definition 5-3

pointer fetch method for fetching multiple rows 5-33

PREPARE statement 5-7

processing of queries 7-4

Q

query
 associating a cursor 3-5
 definition 3-1
 that returns a single row 3-2
 that returns multiple rows 3-3

query types
 full segment 7-3
 simple 7-2

R

- read lock 8-6
 - REAL data type 2-15
 - releasing a locator variable 6-8
 - requirements for compiling and linking ESQL programs
 - issuing the esqlc command 9-3
 - setting environment variables 9-1
 - resetting data types 5-19
 - result set, definition 1-7
 - retrieving rows using a cursor 3-7
 - reusing an SQLDA 5-29
 - ROLLBACK WORK statement
 - description 8-4
 - example 8-4
 - format 8-4
 - rules for
 - coding a Declare Section 2-2
 - extractor qualification 7-4
 - using a host variable 2-26
 - using an indicator variable 2-27
 - variable declarations 2-3
- ## S
- sample program (ESQL) A-1
 - scan, full table 7-1
 - scenarios for dynamic SQL
 - fixed-list SELECTs 5-10
 - non-SELECT with markers 5-6
 - non-SELECT without markers 5-5
 - SELECT using an SQLDA 5-12
 - select list, definition 3-1
 - SELECT statement
 - FOR clause 3-2
 - FROM clause 3-2
 - full segment query 7-3
 - GROUP BY clause 3-2
 - HAVING clause 3-2
 - INTO clause 3-2
 - ORDER BY clause 3-2
 - simple query 7-2
 - WHERE clause 3-2
 - semi-colon delimiter 1-10
 - serializable isolation level 8-5
 - SET CONNECTION statement
 - connection_name argument 2-31
 - description 2-30
 - example 2-31
 - format 2-31
 - setting environment variables for the ESQL precompiler 9-1
 - shared lock 8-6
 - simple query
 - definition 7-2
 - example 7-2
 - processing 7-4
 - SMALLINT data type 2-16
 - SQL format conventions xiv
 - SQL in C, guidelines for embedding 1-8
 - SQL statements
 - continuing on another line 1-10
 - dynamic 5-1

Index

S

- that control cursors
 - CLOSE 3-5
 - DECLARE 3-4
 - FETCH 3-5
 - OPEN 3-4
- SQL syntax
 - braces { } xiv
 - commas , xiv
 - ellipsis points ... xiv
 - lowercase terms xiv
 - uppercase terms xiv
 - vertical bar | xiv
- SQLCA
 - components
 - sqlcabc 4-2
 - sqlcaid 4-2
 - sqlcode 4-2
 - sqlerrd 4-3
 - sqlerrm 4-2
 - sqlerrml 4-2
 - sqlerrp 4-2
 - sqlstate 4-3
 - sqlwarn 4-3
 - description 1-7
 - purpose 4-1
 - structure 4-2
 - WHENEVER statement 4-6
- sqlcabc component of SQLCA structure 4-2
- sqlcaid component of SQLCA structure 4-2
- sqlcode
 - checking the value of 4-4
 - component of SQLCA structure 4-2
- SQLDA
 - allocating 5-21
 - allocating SQLDA buffers for variables 5-27
 - array structure 5-18
 - calculating buffer size 5-27
 - checking the size 5-24
 - components
 - sqldaid 5-15
 - sqldfmod 5-15
 - sqldnrow 5-15
 - sqldnvar 5-15
 - sqldrsv1 5-15
 - sqldsize 5-15
 - sqldvar 5-15
 - sqldvnl 5-15
 - sqldvrsn 5-15
 - sqlv_types 5-18
 - sqlvln32 5-18
 - description 1-4, 5-12
 - freeing 5-24
 - initializing buffer pointers 5-28
 - status values 5-29
 - structure definition 5-14
 - using for array fetches 5-32
- sqldaid SQLDA component 5-15
- sqldfmod SQLDA component 5-15
- sqldnrow SQLDA component 5-15
- sqldnvar SQLDA component 5-15
- sqldrsv1 SQLDA component 5-15
- sqldsize SQLDA component 5-15
- sqldvar SQLDA component 5-15
- sqldvnl SQLDA component 5-15
- sqldvrsn SQLDA component 5-15
- sqlerrd component of SQLCA structure 4-3, 7-5
- sqlerrm component of SQLCA structure 4-2
- sqlerrml component of SQLCA structure 4-2
- sqlerrp component of SQLCA structure 4-2

- sqlstate component of SQLCA structure 4-3
- SQL-style comments 1-8
- sqlv_types component of SQLDA structure 5-18
- sqlvbl32 in tpe_sqlvar 5-16
- sqlvdata in tpe_sqlvar 5-17
- sqlvind in tpe_sqlvar 5-17
- sqlvisnl in tpe_sqlvar 5-17
- sqlvlenp in tpe_sqlvar 5-16
- sqlvln32 component of SQLDA structure 5-18
- sqlvln32 in tpe_sqlvar 5-16
- sqlvname in tpe_sqlvar 5-18
- sqlvprec in tpe_sqlvar 5-17
- sqlvrsv1 in tpe_sqlvar 5-18
- sqlvrsv2 in tpe_sqlvar 5-18
- sqlvscal in tpe_sqlvar 5-17
- sqlvtype in tpe_sqlvar 5-17
- sqlwarn
 - checking for warnings in the SQLCA 4-5
 - component of SQLCA structure 4-3
- standard method for fetching multiple rows 5-32
- starting a transaction 8-2
- statement labels
 - example 1-11
 - format 1-11
- statements
 - CLOSE 3-9
 - COMMIT WORK 8-3
 - CONNECT 2-29
 - DECLARE 3-5
 - DISCONNECT 2-31
 - EXECUTE 5-8
 - EXECUTE IMMEDIATE 5-6
 - FETCH 3-7
 - FREE LOCATOR 6-8
 - OPEN 3-6
 - PREPARE 5-7
 - ROLLBACK WORK 8-4
 - SELECT 3-2, 7-2, 7-3
 - SET CONNECTION 2-30
 - VALUES INTO 6-7
 - WHENEVER 1-8
- static SQL
 - definition 1-3
 - when to use 1-4
- status values, SQLDA 5-29
- STHROOT environment variable 9-2
- StorHouse data types
 - list 2-6
 - mapping to C language types 2-4
- StorHouse extractor, definition 7-1
- StorHouse, description ix
- StorHouse/RM API functions
 - tpe_compare_data B-3
 - tpe_conv_data B-1
- StorHouse/RM, description x
- StorHouse/SM, description ix
- storing dynamic SQL as a character string 5-2
- storing information in an SQLDA using
 - DESCRIBE SELECT LIST FOR 5-13
- substitution marker, definition 5-3

T

TIME data type 2-16

TIMESTAMP data type 2-17

tips for writing ESQL applications C-1

tpe_blob_file_t C structure 2-8

tpe_blob_t C structure 2-8

tpe_clob_file_t C structure 2-10

tpe_compare_data API function B-1
description B-3
syntax B-3

tpe_conv_data API function B-1
description B-1
syntax B-1

tpe_da_alloc function 5-21

tpe_da_alloc_varnames function 5-22

tpe_da_free function 5-24

tpe_da_getbsize function 5-27

tpe_da_setentry function 5-24

tpe_da_setptrs function 5-28

tpe_da_setup function 5-22

tpe_date_t C structure 2-12

tpe_num_t C structure 2-14

tpe_sqlda C structure 5-14

tpe_sqlvar C structure 5-16

tpe_time_t C structure 2-16

tpe_timestamp_t C structure 2-18, 2-19

transaction

definition 8-1

ending 8-3

starting 8-2

transaction management statements 1-4

U

uint16_t data type 2-4

uint32_t data type 2-4

uint64_t data type 2-4

uint8_t data type 2-4

uppercase in SQL syntax xiv

using a cursor with host variable arrays 3-9

V

VALUES INTO statement 6-7

VARBINARY data type 2-19

VARCHAR data type 2-19

variable declarations, rules 2-3

vertical bar in SQL syntax xiv

W

WHENEVER statement 1-8

action_sp argument 4-7

description 4-6

example 4-8

exception_sp argument 4-7

format 4-6

scope 4-7

WHERE clause in SELECT statement 3-2, 7-3