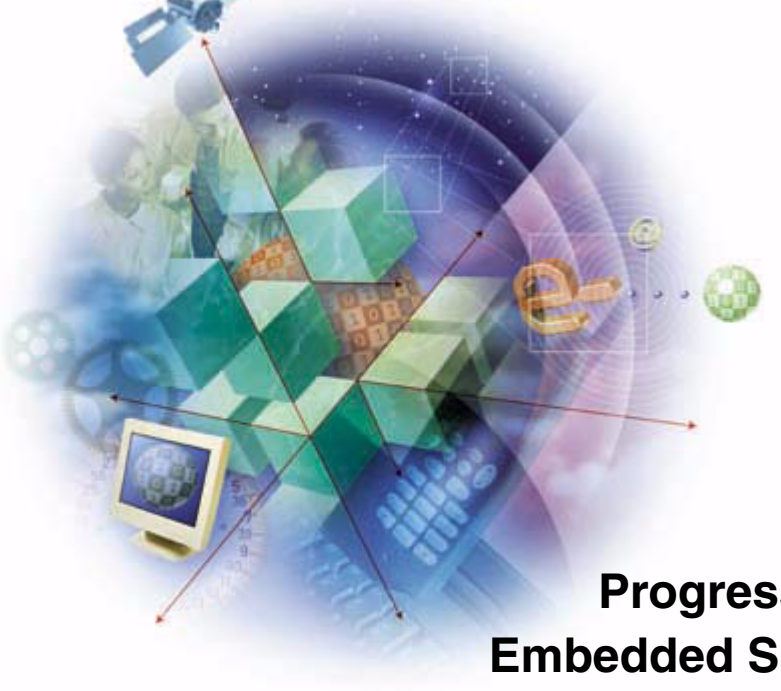

Progress Embedded SQL-92 Guide and Reference

© 2001 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Progress, Progress Results, Provision and WebSpeed are registered trademarks of Progress Software Corporation in the United States and other countries. Apptivity, AppServer, ProVision Plus, SmartObjects, IntelliStream, and other Progress product names are trademarks of Progress Software Corporation.

SonicMQ is a trademark of Sonic Software Corporation in the United States and other countries.

Progress Software Corporation acknowledges the use of Raster Imaging Technology copyrighted by Snowbound Software 1993-1997 and the IBM XML Parser for Java Edition.

© IBM Corporation 1998-1999. All rights reserved. U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Progress is a registered trademark of Progress Software Corporation and is used by IBM Corporation in the mark Progress/400 under license. Progress/400 AND 400® are trademarks of IBM Corporation and are used by Progress Software Corporation under license.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Any other trademarks and/or service marks contained herein are the property of their respective owners.

.

May 2001



Product Code: 4521
Item Number: 81089;9.1C

Contents

Preface . . . . . **xiii**
Purpose xiii
Audience xiii
Organization of This Manual xiii
How to Use This Manual xv
Typographical Conventions xvi
Syntax Notation xvii
Other Useful Documentation xx
Getting Started xx
Development Tools xxi
Database xxi
SQL-92 xxi
Reference xxi
SQL-92 Reference. xxii

1. Introduction **1–1**
1.1 Embedding SQL-92 Statements in a C Language Program 1–2
1.2 Advantages of Using ESQL 1–3
1.3 Program Components of an ESQL Application 1–3
1.4 Embedded SQL Precompiler ESQLC 1–4

2. Using the ESQLC Command **2–1**
2.1 Overview of the ESQL Precompiler 2–2
2.2 Filenames for ESQL Programs 2–2
2.3 ESQLC Command 2–2
2.3.1 Invoking the ESQLC Command 2–3
2.3.2 Platform-specific C Compiler Commands 2–4
2.4 Building an ESQL Application Program Executable 2–6
2.5 Examples Illustrating Precompiler Options 2–7

2.5.1 Use the +V Option for Verbose Output 2–7
2.5.2 Use the +K Option to Keep Intermediate Files 2–8
2.5.3 Use the +G Option to Insert Debug Statements 2–9

3. ESQL-92 Program Structure 3–1
3.1 Defining Variables in a DECLARE SECTION 3–2
3.2 ESQL Executable Statements 3–3
3.2.1 Using Host Variables. 3–3
3.2.2 Using Indicator Variables 3–5
3.2.3 Using Indicator Variables with INPUT Host Variables. 3–7
3.2.4 Using Indicator Variables with OUTPUT Host Variables. 3–8
3.2.5 Limitations of the DECLARE SECTION 3–11
3.3 Types of Executable Statements 3–11
3.4 Error Handling in ESQL Programs 3–11
3.4.1 SQL Communications Area (SQLCA) 3–12
3.4.2 Using the WHENEVER Statement 3–12
3.5 Transaction Management Statements 3–13

4. Connection Management in ESQL-92 4–1
4.1 Introduction 4–2
4.2 CONNECT Statement 4–2
4.2.1 CONNECT Using a Connection Name 4–3
4.2.2 CONNECT by DEFAULT 4–5
4.2.3 CONNECT to a Remote Database 4–5
4.3 SET CONNECTION Statement 4–5
4.4 DISCONNECT Statement 4–6

5. SQL-92 Data Definition Statements 5–1
5.1 CREATE TABLE and DROP TABLE Statements 5–2
5.1.1 CREATE TABLE 5–2
5.1.2 DROP TABLE 5–4
5.2 CREATE INDEX and DROP INDEX Statements 5–5
5.2.1 CREATE INDEX 5–5
5.2.2 DROP INDEX 5–6
5.3 CREATE VIEW and DROP VIEW Statements 5–7
5.3.1 CREATE VIEW 5–7
5.3.2 DROP VIEW 5–8
5.4 Integrity Constraints 5–9
5.4.1 Need for Integrity Constraints 5–9
5.4.2 Types of Integrity Constraints 5–10
5.4.3 Check Constraints 5–11
5.4.4 Primary Keys 5–14

5.4.5 Candidate Keys 5–16
5.4.6 Referential Constraints 5–18
5.4.7 Handling Cycles in Referential Integrity 5–21
5.5 DDL Statements in Long-running Transactions 5–24

6. SQL-92 Data Manipulation Statements 6–1
6.1 Using DML Statements 6–2
6.1.1 INSERT Rows into a Table 6–2
6.1.2 DELETE Rows from a Table 6–5
6.1.3 UPDATE Rows in a Table 6–7
6.2 INPUT Host Variables in DML Statements 6–8

7. Query Statements 7–1
7.1 Elements of a Query 7–2
7.1.1 SELECT Statement Clauses 7–2
7.1.2 INPUT Host Variables in Query Statements 7–3
7.1.3 OUTPUT Host Variables in Query Statements 7–3
7.2 Queries Returning a Single Row 7–4
7.3 Queries Returning Multiple Rows 7–5
7.3.1 Introduction to Cursors 7–5
7.3.2 Associating a Cursor with a Query 7–5
7.3.3 OPEN a Cursor 7–6
7.3.4 FETCH Rows Using a Cursor 7–7
7.3.5 CLOSE a Cursor 7–9
7.3.6 UPDATE or DELETE the Current Row 7–9
7.4 Array Fetches 7–11

8. Handling NULL Values in ESQL-92 8–1
8.1 Introduction 8–2
8.2 INSERT NULL Values 8–2
8.2.1 INSERT Null Values by DEFAULT 8–3
8.2.2 Use the NULL Keyword 8–4
8.2.3 Use Indicator Variables 8–5
8.3 UPDATE with NULL Values 8–6
8.4 RETRIEVE NULL Values 8–6
8.4.1 Using Indicator Variables 8–7
8.4.2 Using the NVL Scalar Function 8–7
8.5 NULL Values in Expressions 8–8
8.6 NULL Values in the WHERE Clause 8–8
8.7 NULL Values in a GROUP BY Clause 8–9
8.8 NULL Values in an ORDER BY Clause 8–9
8.9 NULL Values in Scalar Functions 8–10
8.10 NULL Values in Aggregate Functions 8–10

9. Handling Errors in ESQL-92 9–1
9.1 Using an SQLCA for Error Handling 9–2
9.2 The SQLCA Structure 9–2
9.2.1 Using an SQLCA to Check for Errors 9–5
9.2.2 Using an SQLCA to Check for Warnings 9–8
9.3 Using the WHENEVER Statement for Error Handling 9–10
9.3.1 Branching Under Exceptions. 9–10
9.3.2 Scope of the WHENEVER Statement. 9–13
9.4 Using Indicator Variables for Error Handling 9–14

10. Dynamic SQL Management in ESQL-92 10–1
10.1 Introduction 10–2
10.2 PREPARE Statement 10–2
10.3 EXECUTE Statement 10–4
10.4 EXECUTE IMMEDIATE Statement 10–5
10.5 SELECT Statement 10–6
10.5.1 OPEN a Cursor 10–6
10.5.2 FETCH Rows Using a Cursor 10–6
10.5.3 CLOSE a Cursor 10–8
10.6 SQL Descriptor Area (SQLDA) 10–8
10.6.1 When to Use an SQLDA 10–8
10.6.2 SQLDA Structure 10–9
10.6.3 Guidelines for Using an SQLDA 10–14
10.6.4 Allocating an SQLDA 10–15
10.6.5 Using an SQLDA for Array Fetches 10–16
10.6.6 Freeing an SQLDA 10–19
10.7 DESCRIBE Statement 10–20
10.8 SQLDA for Input Variables 10–22
10.9 SQLDA for Output Variables 10–24

11. Transaction Management in ESQL-92 11–1
11.1 Introduction to Transactions 11–2
11.1.1 Starting a Transaction 11–2
11.1.2 COMMIT Statement 11–4
11.1.3 ROLLBACK Statement 11–4
11.2 Transaction Isolation Levels 11–5
11.3 Locking and Transactions 11–7
11.4 Abnormal Termination of an ESQL Application Program 11–9
11.4.1 Forced Rollback of a Transaction 11–9
11.4.2 Interrupting the Execution of an SQL Statement. 11–9

12. Guidelines for Developing ESQL-92 Applications 12–1
12.1 Coding Guidelines 12–2
12.2 Using SQL for Computation 12–4

12.3 Using SQL for Condition Evaluation 12–6
12.4 Using Indicator Variables 12–6
12.5 Using Scalar Functions 12–7
12.6 Using Static and Dynamic Statements 12–8

13. Handling Data Types in ESQL-92 13–1
13.1 Introduction 13–2
13.2 Character Data Types 13–2
13.2.1 CHARACTER Data Type 13–2
13.2.2 VARCHAR Data Type 13–3
13.3 Exact Numeric Data Types 13–4
13.3.1 INTEGER Data Type 13–4
13.3.2 SMALLINT Data Type 13–4
13.3.3 TINYINT Data Type 13–5
13.3.4 NUMERIC or DECIMAL Data Type 13–5
13.4 Approximate Numeric Data Types 13–7
13.4.1 REAL Data Type 13–7
13.4.2 FLOAT Data Type 13–7
13.4.3 DOUBLE PRECISION Data Type 13–8
13.5 Date-time Data Types 13–8
13.5.1 DATE Data Type 13–8
13.5.2 TIME Data Type 13–9
13.5.3 TIMESTAMP Data Type 13–10
13.6 Bit String Data Types 13–10
13.6.1 BIT Data Type 13–10
13.6.2 BINARY Data Type 13–10
13.6.3 VARBINARY Data Type 13–11
13.7 Data Conversion 13–11
13.7.1 Implicit Data Conversion 13–11
13.7.2 Explicit Data Conversion Using *tpe_conv_data( )* 13–11
13.7.3 Explicit Data Conversion Using Scalar Functions 13–14
13.7.4 Explicit Data Conversion Using SQL and *syscalctable* 13–15
13.8 Data Comparison 13–16
13.8.1 Explicit Data Comparison Using *tpe_compare_data( )* 13–16
13.8.2 Explicit Data Comparison Using SQL and *syscalctable* 13–19

14. Embedded SQL-92 Reference 14–1
BEGIN-END DECLARE SECTION 14–3
CLOSE Statement 14–5
DECLARE CURSOR Statement 14–7
DESCRIBE Statements 14–10
DESCRIBE BIND VARIABLES Statement 14–11
DESCRIBE SELECT LIST Statement 14–13
EXEC SQL Delimiter 14–15

EXECUTE Statement 14–17
EXECUTE IMMEDIATE Statement 14–19
FETCH Statement 14–21
OPEN Statement 14–24
PREPARE Statement 14–27
Query Expressions 14–31
Search Conditions 14–33
SELECT Statement for a Single Row 14–35
Type Specifications for Host Language Types 14–38
Type Specifications for Static Array Types 14–40
WHENEVER Statement 14–43

A. ESQL-92 Sample Programs A–1
1StatUpd.pc A–2
2StatSel.pc A–8
3DynUpd.pc A–14
4DynSel.pc A–20

B. Compile, Link, and Run-time Requirements B–1
B.1 Requirements for All Platforms B–2
B.2 Compile, Link, and Run-time Requirements by Platform B–2

Glossary Glossary–1

Index Index–1

Figures

Figure 2–1: Embedded SQL-92 and C source => Executable Application Program 2–6

Tables

Table 3–1:	Host Variables in an INPUT Example	3–2
Table 3–2:	Indicator Variable Values and Their Meanings	3–6
Table 3–3:	Host Variables in an OUTPUT Example	3–9
Table 12–1:	Indicator Variable Values and Their Meanings	12–7
Table 13–1:	Values Returned by *tpe_compare_data()*	13–17
Table A–1:	Sample Programs	A–1
Table B–1:	Compile, Link, and Run-time Requirements by Platform	B–2

Procedures

1StatUpd.pc A–2
2StatSel.pc A–8
3DynUpd.pc A–14
4DynSel.pc A–20

Preface

Purpose

The manual describes the syntax and semantics of the Progress Embedded SQL-92 language statements.

Audience

This manual is intended for application developers writing database applications using the Progress Embedded SQL-92 environment. See also the *Progress SQL-92 Guide and Reference*.

Organization of This Manual

Chapter 1, "Introduction"

Introduces the concept of embedding SQL statements in a host language and the advantages of using embedded SQL.

Chapter 2, "Using the ESQLC Command"

Provides a description of how to invoke the ESQLC command, including ESQLC command syntax and options. Lists platform-specific requirements.

Chapter 3, "ESQL-92 Program Structure"

Describes the overall structure of an ESQL program, including the DECLARE SECTION, host variables, and types of ESQL executable statements.

Chapter 4, "Connection Management in ESQL-92"

Describes connection management statements in ESQL.

Chapter 5, “SQL-92 Data Definition Statements”

Introduces the data definition statements (DDL) that can be used in an ESQL program and gives a description of integrity constraints that can be used in DDL statements.

Chapter 6, “SQL-92 Data Manipulation Statements”

Describes using Data Manipulation statements (DML) in ESQL programs and discusses the types of DML statements.

Chapter 7, “Query Statements”

Describes SQL queries and their use in ESQL programs. Introduces the SELECT statement and discusses the query statements with host variables.

Chapter 8, “Handling NULL Values in ESQL-92”

Describes how to handle NULL values in ESQL and discusses inserting and retrieving NULL values.

Chapter 9, “Handling Errors in ESQL-92”

Describes handling errors in ESQL and discusses the use of the SQLCA, the WHENEVER statement, and indicator variables for error handling.

Chapter 10, “Dynamic SQL Management in ESQL-92”

Describes the combinations of steps required to process an SQL-92 statement using dynamic SQL.

Chapter 11, “Transaction Management in ESQL-92”

Describes transaction management in ESQL and explains how to start and end a transaction, set transaction isolation levels, and manage locking in transactions.

Chapter 12, “Guidelines for Developing ESQL-92 Applications”

Contains some general guidelines for developing ESQL applications.

Chapter 13, “Handling Data Types in ESQL-92”

Describes the data types that are available in an ESQL application program and discusses mechanisms for data conversions and data comparisons.

Chapter 14, "Embedded SQL-92 Reference"

Provides syntax, examples, and reference information on statements used for processing embedded SQL-92 statements.

Appendix A, "ESQL-92 Sample Programs"

Complete code listings of ESQL-92 sample programs.

Program Name	Sample Program Illustrates
"1StatUpd.pc"	Static SQL-92 UPDATE Statement
"2StatSel.pc"	Static SQL-92 SELECT Statement
"3DynUpd.pc"	Dynamic SQL-92 non-SELECT Statement
"4DynSel.pc"	Dynamic SQL-92 SELECT Statement

Appendix B, "Compile, Link, and Run-time Requirements"

This appendix lists compile, link, and run time requirements for using the Embedded SQL-92 precompiler for C.

"Glossary"

How to Use This Manual

Use this manual as a reference for Progress Embedded SQL-92 and as a guide to building and compiling Progress Embedded SQL-92 applications. This manual is not a reference for the C programming language. For information on supported SQL language elements and statements see the *Progress SQL-92 Guide and Reference*.

Typographical Conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
  - Commands or characters that the user types
  - That a word carries particular weight or emphasis
- *Italic typeface* indicates:
  - Progress variable information that the user supplies
  - New terms
  - Titles of complete publications
- `Monospaced typeface` indicates:
  - Code examples
  - System output
  - Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.

  END-ERROR, **GET**, **GO**
  ALT, **CTRL**, **SPACEBAR**, **TAB**

- When you have to press a combination of keys, they are joined by a dash. You press and hold down the first key, then press the second key.

  CTRL-X

- When you have to press and release one key, then press another key, the key names are separated with a space.

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each SQL-92 component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in an expression or a statement.

  In this example, GRANT, RESOURCE, DBA, and TO are keywords:

  SYNTAX

  ```
  GRANT { RESOURCE, DBA } TO user_name [, user_name ] ... ;
  ```

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the GRANT statement you must supply at least one *user_name*.

- Square brackets (**[ ]**) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

  In this syntax example the first *user_name* is required, and additional *user_name* specifications are optional:

  SYNTAX

  ```
  GRANT { RESOURCE, DBA } TO user_name [, user_name ] ... ;
  ```

  NOTE: The ellipsis (**. . .**) indicates repetition, as shown in a following description.

- Braces (**{ }**) around an item indicate that the item, or a choice of one of the enclosed items, is required.

  In the GRANT example, you must specify either RESOURCE or DBA or both, and at least one *user_name*. Any *user_name* specifications after the first are optional:

  SYNTAX

  ```
  GRANT { RESOURCE, DBA } TO user_name [, user_name ] ... ;
  ```

  In some cases, braces are not a syntax notation, but part of the language.

For example, this excerpt from an ODBC application invokes a stored procedure using the ODBC syntax { call *procedure_name* (*param*) }, where braces and parentheses are part of the language:

SYNTAX

```
proc1( param, "{ call proc2 (param) }", param);
```

- A vertical bar (**|**) indicates a choice. In the CREATE SYNONYM example you must specify a *table_name*, *view_name*, or *synonym* but you can only choose one. Note that in all SQL-92 syntax, if you specify the optional *owner_name* qualifier, there must not be a blank between the period separator and the *table_name* or *view_name* or *synonym*:

SYNTAX

```
CREATE [ PUBLIC ] SYNONYM synonym
 FOR [ owner_name.]{table_name |view_name |synonym } ;
```

In the DELETE FROM example, you must specify one of *table_name* or *view_name*:

SYNTAX

```
DELETE FROM [ owner_name.]{table_name |view_name }
  [ WHERE search_condition ] ;
```

- Ellipses (**. . .**) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

In the ORDER BY example, you must include one expression *(expr)* or column position *(posn)*, and you can optionally specify the sort order as ascending (ASC) or descending (DESC). You can specify additional expressions or column positions for sorting within a sorted result set. The SQL-92 engine orders the rows on the basis of the first *expr* or *posn*. If the values are the same, the second *expr* or *posn* is used in the ordering:

SYNTAX

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

In the GRANT example, you must include one *user_name*, but you can optionally include more. Note that each subsequent *user_name* must be preceded by a comma:

SYNTAX

```
GRANT { RESOURCE, DBA } TO user_name [, user_name ] ... ;
```

- In many examples, the syntax is too long to place in one horizontal row. In such cases, **optiona**l items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

  In this example, CREATE VIEW is followed by several optional items:

  SYNTAX

```
CREATE VIEW [ owner_name.]view_name
  [ ( column_name [, column_name ] ... ) ]
  AS [ ( ] query_expression [ ) ] [ WITH CHECK OPTION ] ;
```

Other Useful Documentation

This section lists Progress Software Corporation documentation that you might find useful. Unless otherwise specified, these manuals support both Windows and Character platforms and are provided in electronic documentation format on CD-ROM.

Getting Started

Progress Electronic Documentation Installation and Configuration Guide (Hard copy only)

A booklet that describes how to install the Progress EDOC viewer and collection on UNIX and Windows.

Progress Installation and Configuration Guide Version 9 for UNIX

A manual that describes how to install and set up Progress Version 9.1 for the UNIX operating system.

Progress Installation and Configuration Guide Version 9 for Windows

A manual that describes how to install and set up Progress Version 9.1 for all supported Windows and Citrix MetaFrame operating systems.

Progress Version 9 Product Update Bulletin

A guide that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Progress Master Glossary for Windows and *Progress Master Glossary for Character* (EDOC only)

Platform-specific master glossaries for the Progress documentation set. These books are in electronic format only.

Progress Master Index and Glossary for Windows and *Progress Master Index and Glossary for Character* (Hard copy only)

Platform-specific master indexes and glossaries for the Progress hard-copy documentation set.

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup commands and parameters in alphabetical order.

Development Tools

Progress Basic Database Tools (Character only; information for Windows is in online help)

A guide for the Progress Database Administration tools, such as the Data Dictionary.

Database

Progress Database Design Guide

A guide that uses a sample database and the Progress Data Dictionary to illustrate the fundamental principles of relational database design. Topics include relationships, normalization, indexing, and database triggers.

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures. The procedures allow you to create and maintain your Progress databases and manage their performance.

SQL-92

Progress JDBC Driver Guide

A guide to the Java Database Connectivity (JDBC) interface and the Progress SQL-92 JDBC driver. It describes how to set up and use the driver and details the driver's support for the JDBC interface.

Progress ODBC Driver Guide

A guide to the ODBC interface and the Progress SQL-92 ODBC driver. It describes how to set up and use the driver and details the driver's support for the ODBC interface.

Progress SQL-92 Guide and Reference

A user guide and reference for programmers who use Progress SQL-92. It includes information on all supported SQL-92 statements, SQL-92 Data Manipulation Language components, SQL-92 Data Definition Language components, and Progress functions. The guide describes how to use the Progress SQL-92 Java classes and how to create and use Java stored procedures and triggers.

Reference

Pocket Progress (Hard copy only)

A reference that lets you quickly look up information about the Progress language or programming environment.

SQL-92 Reference

These are non-Progress resources available from your technical bookseller.

A Guide to the SQL Standard

Date, C.J., with Hugh Darwen. 1997. Reading, MA: Addison Wesley.

Understanding the New SQL: A Complete Guide

Melton, Jim (Digital Equipment Corporation) and Alan R. Simon. 1993. San Francisco: Morgan Kaufmann Publishers.

1

Introduction

This chapter provides an introduction to Embedded SQL-92 (ESQL). It discusses the concept of embedding SQL statements in a host language and the advantages of using ESQL. It also discusses the components of an ESQL program and gives an introduction to the Progress ESQL-92 precompiler.

1.1 Embedding SQL-92 Statements in a C Language Program

SQL is a non-procedural language that uses statements to define, manipulate, and control data in a relational database. The Progress SQL-92 application development environment allows development of applications using the Embedded SQL tool. ESQL allows you to embed SQL-92 statements in the widely used C programming language. In this context, the C Language is referred to as the host language. Embedding SQL statements in a host language allows for development of applications that are more powerful and flexible than applications developed in either the host language or SQL alone. The Progress embedded SQL-92 interface in C is referred to as ESQLC.

C Language compilers do not recognize SQL statements and ESQL application programs containing SQL statements. It is necessary to translate these statements to the equivalent C Language code to build the program executable. The ESQL precompiler performs this translation.

To facilitate recognition of the statements the ESQL precompiler is to process, you begin the statements with the prefix *EXEC SQL*. These prefixed statements are referred to as *ESQL constructs*. The ESQL constructs are outlined in following sections.

NOTE: See Chapter 3, "ESQL-92 Program Structure," for a complete description of each ESQL construct.

EXAMPLE

The following code fragment is an overview of how SQL statements can be embedded in C:

```
EXEC SQL BEGIN DECLARE SECTION ;
     long order_no_v ;
     char order_date_v [10] ;
     char product_v [5] ;
     long qty_v ;
EXEC SQL END DECLARE SECTION ;

/*
** C Language code to determine values for host variables.
** Typically you retrieve values through a user interface.
*/
order_no_v = 1001 ;
strcpy (order_date_v, "02/02/1999") ;
strcpy (product_v, "COG") ;
qty_v = 10000 ;
printf ("Registering the order\n") ;

EXEC SQL
     INSERT INTO orders
          (order_no, order_date, product, qty)
     VALUES
          (:order_no_v, :order_date_v, :product_v, :qty_v) ;
```

1.2 Advantages of Using ESQL

The advantages of using ESQL as an application development tool are:

- Allows the application developer to combine the flexibility of a host language and the power of SQL for data access
- Reduces the source code size and complexity of an application without reducing the implementation flexibility

1.3 Program Components of an ESQL Application

There are two categories of SQL statements in an ESQL program:

- ESQL DECLARE statements
- ESQL executable statements

Use the ESQL DECLARE statements to declare variables for use in ESQL constructs. The DECLARE statements you can use in an ESQL application program are:

- EXEC SQL BEGIN DECLARE SECTION
- EXEC SQL END DECLARE SECTION
- EXEC SQL WHENEVER
- EXEC SQL DECLARE CURSOR

ESQL executable statements form the body of an ESQL program. These statements are called executable statements because they result in the execution of instructions against a particular database at run time.

The executable statements include Data Manipulation Language (DML) statements, Data Definition Language (DDL) statements, and Data Control Language (DCL) statements.

1.4 Embedded SQL Precompiler ESQLC

The ESQL precompiler ESQLC is a tool for building a database application in the Progress SQL-92 development environment. The ESQL precompiler translates embedded SQL statements into host language statements that include ESQL library calls. The output is a C Language program. You use one or more C programs to build an application program executable.

These are the general steps involved in building an ESQL program executable:

1. Develop ESQL source programs.
2. Use the ESQL precompiler to precompile each of the ESQL programs that constitute an application and generate the corresponding C source file.
3. Use a standard C compiler to create the object file(s) from the C source file(s).
4. Link the object files with the ESQL libraries to build the application executable.

NOTE: For more information on the ESQL precompiler see Chapter 2, "Using the ESQLC Command."

2

Using the ESQLC Command

This chapter provides a description of how to invoke the ESQLC command. Specifically, it discusses:

- Filename conventions
- ESQLC command syntax
- Building an application executable from an ESQLC source file

2.1 Overview of the ESQL Precompiler

The ESQL precompiler interprets ESQL source files and generates C Language source files. Subsequently a C compiler compiles and links the source files into an executable file. The compile and link steps are automatic, unless you enter options to override these steps when you invoke the ESQLC command. For ease of use, the ESQL precompiler also accepts C source files and object files, passing them to the compiler and linker. You must ensure that the location of the ESQL precompiler is in your path before you invoke the precompiler from the command prompt. The location of the ESQL precompiler is the `$DLC/bin` install directory. The ESQLC precompiler also provides a mechanism for passing compiler options to the C Language compiler. See the "Platform-specific C Compiler Commands" section.

The ESQL precompiler for C has specific environment variable and compiler requirements. These vary across platforms. The run-time environment requires dynamic load library routines that are available in C++ development environments. In most cases, a C Language development environment is not sufficient.

See Appendix B, "Compile, Link, and Run-time Requirements," for information on requirements, grouped by platform.

2.2 Filenames for ESQL Programs

A database application developed using ESQL in the Progress SQL-92 environment is comprised of a set of source files. These source files implement the application logic using embedded SQL constructs and the C Language. If a file contains embedded SQL constructs and you are processing the file with the ESQL precompiler, you must name it using the file suffix `.pc`. For example, an ESQL application program developed for order processing and containing embedded SQL might have the filename `orderproc.pc`.

2.3 ESQLC Command

The ESQL precompiler accepts options and a list of input files at the command line. If you enter the ESQLC command with no options, the precompiler returns a formatted usage summary. The display shows the required syntax and lists all of the available options.

EXAMPLE

The following example illustrates the usage display returned if you invoke the ESQLC command with no options and no filenames:

```
> esqlc

                    Progress/esqlc Version 09.1A

               Progress Software Corporation   (c) 1988-99
Usage: esqlc [options] files
Options:                                          (default values)
                  : Produce client.exe by default   ()
        -o        : Produce client.exe by default   (Link)
        -c        : Produce .o file w/o linking     (Compile)
        +T        : Produce .c file w/o compiling   (Preprocess)
        +P        : Invoke CPP to preprocess the .pc files  (no)
        +L        : Generate the line numbers       (no)
        +V        : Verbose mode on                 (off)
        +K        : Keep intermediate files         (Remove all)
        +G        : Insert debugging code and +V +K (None)
        -[Compiler Options]  : Passes [Compiler Options] to compiler
               (UNIX only)
        +[Compiler Options]  : Passes [Compiler Options] to compiler
               (Win32 only)
-------------------------------------------------------------------------
```

2.3.1 Invoking the ESQLC Command

This is the syntax for invoking the ESQLC command:

SYNTAX

```
ESQLC [ option_list ] file_name_list
```

option_list

Options that are recognized and processed by the ESQL precompiler have the plus sign (+) prefix to avoid clashes with options supported by a C Language compiler and linker. Available options you can specify in the *option_list* are:

SYNTAX

```
+T | +K | +P | +L | +V | +G
```

+T

Directs the precompiler to interpret the embedded SQL source code in a file with the `.pc` suffix, and generate C source code. There is no further processing of the generated C source code. When you specify +T, the precompiler sets the +K option, which retains any intermediate files.

+K

Directs the precompiler to retain any intermediate files.

+P

Directs the precompiler to run the C preprocessor on the input file before translating the SQL statements in the input file. This allows you to use #define symbols in ESQL statements.

NOTE: If your application inserts integer values into a database and you specify the +P precompiler option, you must not have leading zeroes in the integers.

+L

Suppresses redefinition of source line numbers in the generated C code to simplify debugging of host-language statements. This facilitates tracing in source code since the line number shown by a debugger corresponds to that in the ESQL source file.

+V

Displays the commands the precompiler invokes as it processes the files. Keeps intermediate C source code and object files. If you omit this option, the precompiler does not keep any intermediate files.

+G

Inserts debug print statements, sets the +V and +K options, and passes the debug directive to the compiler.

2.3.2 Platform-specific C Compiler Commands

When you invoke the ESQLC precompiler, you can include C Language compiler options in the same step. The set of valid compiler options is specific to the compiler you are running. See Appendix B, "Compile, Link, and Run-time Requirements," for environment requirements, grouped by platform.

C Language Compiler Options on UNIX Platforms

To pass compile-time options to the compiler when you invoke the ESQLC precompiler on a UNIX platform, the compiler option must begin with a hyphen (-).

EXAMPLE

In this example, the ESQLC precompiler passes the -f option to the UNIX platform C Language compiler:

```
> esqlc +T -f client.pc
```

C Language Compiler Options on Windows NT Platforms

To pass compile-time options to the compiler when you invoke the ESQLC precompiler on a Windows NT platform, prepend a plus sign and a slash (+/) to the compiler option.

NOTE: Exceptions: On Windows NT the precompiler does recognize the -c option (compile only) and the -o option (rename executable program) and passes these options to the C Language compiler.

EXAMPLE

In this example, the ESQLC precompiler passes the Zi option to the Windows NT platform C Language compiler:

```
> esqlc +T +/Zi client.pc
```

2.4 Building an ESQL Application Program Executable

The ESQL precompiler accepts embedded SQL source files with the filename suffix `.pc` and generates C Language source files. A C compiler and linker then process the C source files, creating the application executable. You can perform these operations in one step; the ESQL precompiler invokes the C compiler, which in turn invokes the linker. Alternatively, you can specify command-line options to perform the operations in multiple steps.

Figure 2–1 illustrates the components and steps you use to process SQL statements embedded in C Language source, creating an executable application program.

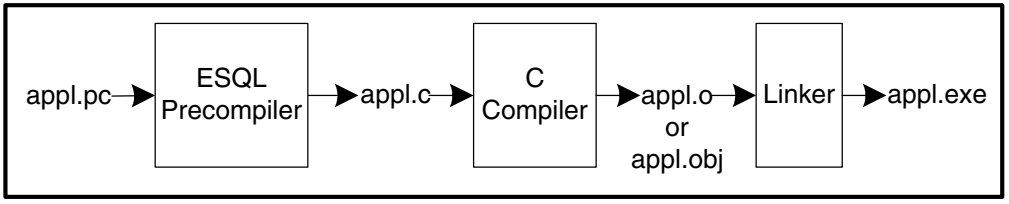

Figure 2–1: Embedded SQL-92 and C source Executable Application Program

The ESQL precompiler option +T directs the precompiler to generate C source code from the ESQL application program source, keep the generated C Language source, and perform no further processing on the output.

The ESQLC command passes any properly formatted compiler options to the compiler. See the documentation for the C compiler in your environment for complete information on valid options. Two C compiler options you might use are listed here. These options are accepted by UNIX and Windows C Language compilers:

- -c

  Directs the C compiler to compile only, and not perform the link step.

- -o *file_name*

  Directs the linker to name the output executable program with the specified name. The default executable program name is `client.exe`. You can assign a different name to the executable program by using the -o option.

You can build the application program `client.exe` in a single invocation of the ESQL precompiler, or you can create it in multiple user-visible steps.

EXAMPLES

The following examples illustrate these operations:

In the first example, the ESQL precompiler processes the embedded SQL source file `client.pc`, generates the C source file `client.c`, compiles the C source into an object file, and links the object file into an executable program named `newname.exe`. The +K option directs the precompiler to keep all intermediate files:

```
> esqlc -o newname +K client.pc
```

The second example completes the same operations as the first example, but in two user-visible steps:

```
> esqlc +T client.pc
> esqlc +K client.c -o newname
```

2.5 Examples Illustrating Precompiler Options

This section provides additional examples of how to use precompiler options.

2.5.1 Use the +V Option for Verbose Output

When you specify +V, the precompiler displays the commands it invokes.

These examples illustrate the difference in output when you do and do not specify the +V (verbose) option. The first invocation specifies +V, and the precompiler displays the commands it invokes:

```
> esqlc +T +V client.pc

                            Progress/esql
                Progress Software Corporation (c) 1999

Translating from FGL ...
D:\PROGRESS\bin\efglc +V +S client.pc client.c

Adding header ...
del  client.c

copy  D:\PROGRESS\fgl326.cxx client.c

del  D:\PROGRESS\fgl326.cxx
done
```

The second ESQLC command performs the same operations, creating a `client.c` output file, but does not display the commands it invokes:

```
> esqlc +T client.pc

                            Progress/esql
                Progress Software Corporation (c) 1999

       1 file(s) copied.
```

2.5.2 Use the +K Option to Keep Intermediate Files

The +K option directs the precompiler and compiler to keep intermediate C source and object files.

This ESQLC command produces the executable application program named `client.exe`. The +K option directs the precompiler and compiler to retain the intermediate C Language source file and object file:

```
> esqlc +K client.pc
```

This is an excerpt from a display of the working directory after executing the ESQLC command with the +K option specified:

```
Directory of D:\dbwrk\esql

02/16/99  12:32p        <DIR>          .
02/16/99  12:32p        <DIR>          ..
02/16/99  10.50a                 1,046 client.exe
02/16/99  10:50a                59,702 client.c
02/16/99  09:30a                   476 client.pc
02/16/99  10:50a                 1,046 client.o

               6 File(s)        62,270 bytes
```

2.5.3 Use the +G Option to Insert Debug Statements

The +G option directs the precompiler to insert debug print statements in the generated C Language source program. When you specify +G, the +V and +K options are set automatically.

This ESQLC command produces a `client.c` source file with debug statements in the source. The precompiler also lists the commands it is executing (+V), and the intermediate files are kept (+K):

```
> esqlc +G client.pc
```

This is an excerpt from a generated program where +G is specified:

```
   .
   .
   .
#define PROGRESS_DEBUG(x) printf("%s\n", x)
#define PROGRESS_DBG(x) printf("%s"), x
   .
   .
   .
```

This is an excerpt from the generated `client.c` program when you do **not** specify the +G option. Verbose mode (+V) is **not** enabled and intermediate files are **not** kept (+K):

```
   .
   .
   .
#define PROGRESS_DEBUG(x)
#define PROGRESS_DBG(x)
   .
   .
   .
```

3

ESQL-92 Program Structure

This chapter describes the overall structure of an embedded SQL (ESQL) source program. Specifically, this chapter discusses the following elements of an ESQL program:

- DECLARE SECTION
- Host variables
- Indicator variables
- Limitations of the DECLARE SECTION
- Types of ESQL executable statements
- Error handling using the SQL Communications Area (SQLCA)
- Transaction management statements

3.1 Defining Variables in a DECLARE SECTION

This section describes the DECLARE SECTION, and illustrates ESQL executable statements in examples. See the "Types of Executable Statements" section for more information on executable statements. This section also includes information on host variables and indicator variables, and how to use them in the DECLARE SECTION, in C Language statements, and in ESQL executable statements.

The ESQL precompiler is a tool that parses ESQL executable statements and translates them into C Language statements. ESQL does not allow the use of C-language defined variables in ESQL executable statements. The precompiler requires a DECLARE SECTION for declarations of any variables you use in ESQL executable statements.

In an ESQL source program, the DECLARE SECTION must precede any other ESQL constructs. Only C Language statements can precede the DECLARE SECTION. The variables defined in the DECLARE SECTION can be declared as local variables or global variables.

NOTE: The data types in the DECLARE section are SQL data types and C/C++ data types. Some, but not all, of the C/C++ data types are defined as synonyms for the corresponding SQL data types. In particular, 'int' is not allowed.

The following code fragment shows how to mark the beginning and end of a DECLARE SECTION and how to define variables:

```
EXEC SQL BEGIN DECLARE SECTION ;
     long order_no_v ;
     char order_date_v [10] ;
     char product_v [5] ;
     long qty_v ;
EXEC SQL END DECLARE SECTION ;
```

The variables you declare in the DECLARE SECTION are called *host variables* and *indicator variables*. Table 3–1 lists the host variables defined in the DECLARE SECTION of this example. The code fragment uses the convention of ending each variable name with "_v" for variable name:

Table 3–1: Host Variables in an INPUT Example *(1 of 2)*

Host Variable Name	Data Type
order_no_v	long integer
order_date_v [10]	character array of size 10 (must include space for the trailing null)

Table 3–1: Host Variables in an INPUT Example *(2 of 2)*

Host Variable Name	Data Type
product_v [5]	character array of size 5 (must include space for the trailing null)
qty_v	long integer

The ESQL precompiler generates the corresponding C Language declarations for these variables. The application program has access to these variables in C Language statements. The following sections provide more detail about host variables, indicator variables, and how to use them.

NOTE: The ESQL precompiler does not check for or generate messages about unused variables in the DECLARE SECTION.

3.2 ESQL Executable Statements

The ESQL constructs that result in the execution of instructions on a specified database at run time are called ESQL executable statements. The executable statements form the body of an ESQL program and are used to access the database.

EXAMPLE

The following example shows an UPDATE executable statement. This UPDATE statement adds 1000 units to the existing quantity (qty) for the row in the orders table where order number (order_no) equals 1244.

```
EXEC SQL
    UPDATE orders
    SET qty = qty + 1000
    WHERE order_no = 1244 ;
```

3.2.1 Using Host Variables

You must declare host variables in the DECLARE SECTION before you can use them in an ESQL executable statement. The data types used to declare host variables generally must be the same types as database types. For example, to declare a host variable of C Language type SHORT, the precompiler allows you to DECLARE the variable as database type SMALLINT.

This is the general format for declaring host variables in the DECLARE SECTION:

SYNTAX

```
EXEC SQL BEGIN DECLARE SECTION ;

    data_type   host_variable_name1 ;
    [ data_type   host_variable_name2 ; ]

EXEC SQL END DECLARE SECTION ;
```

host variable

SYNTAX

```
EXEC SQL
    :host_variable_name
```

NOTE: When you use a host variable in an ESQL executable statement, you must prepend a colon (:) to the variable.

EXAMPLE

The following code fragment shows the use of host variables in an embedded SQL program:

```
EXEC SQL BEGIN DECLARE SECTION ;
     long order_no_v ;
     char order_date_v[10] ;
     char product_v[5] ;
     long qty_v ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
/*
** C Language processing code here, to determine values for host variables
*/
   .
   .
   .
/*
** ESQL executable statement.
** INSERT a row with 4 column values into the orders table.
*/
EXEC SQL
     INSERT INTO orders (order_no, order_date, product, qty)
     VALUES (:order_no_v, :order_date_v, :product_v, :qty_v) ;
```

The following list provides a summary of how and when to use host variables:

- Explicitly DECLARE host variables in the DECLARE SECTION.
- Prepend host variables with a colon (:) when you use them in an ESQL statement.
- Do not prepend host variables with a colon in the DECLARE SECTION or when you use them in a C Language statement.
- Name a host variable with a name that is **not** an SQL reserved word.
- Use a host variable in an ESQL statement only where a constant is valid.
- You can associate an indicator variable with a host variable.

3.2.2 Using Indicator Variables

Indicator variables are optional variables you can use to handle NULL values. Each indicator variable is associated with one host variable. Indicator variables should be of C Language data type LONG or database data type INTEGER (a 32-bit signed integer). The general format for using an indicator variable in an ESQL program is:

SYNTAX

```
:host_variable_name [ INDICATOR ] :indicator_variable_name
```

NOTE: An indicator variable is always used with a host variable and you must prepend it with a colon (:) when you use it in an ESQL executable statement.

`INDICATOR`

The `INDICATOR` keyword is not required. Using the `INDICATOR` keyword clarifies the intent, especially if the names for the host variable and indicator variable do not follow a recognizable convention.

Table 3–2 lists the valid values for indicator variables and describes the meaning for each value.

Table 3–2: Indicator Variable Values and Their Meanings

Value	Meaning of Indicator Variable Values
0	The associated host variable contains a non NULL value. If set by SQL in a fetch operation, 0 also indicates that the value in the host variable has **not** been truncated.
- 1	The returned value in an output operation is NULL and there is no value in the associated host variable. The value of the associated host variable for an UPDATE or INSERT operation is NULL.
> 0	The host variable size was too small to contain the returned value in an output operation, and the returned value in the host variable has been truncated. The indicator variable itself contains the > 0 length of the returned value before it was truncated. Only CHAR/VARCHAR data is truncated. For other data types, no data will be returned when the allocated area is not big enough.

This list provides a summary of how and when to use indicator variables:

- Explicitly DECLARE indicator variables in the DECLARE SECTION, and do not prepend a colon (:) in this section.
- DECLARE indicator variables as C Language data type LONG or database data type INTEGER (a 32-bit signed integer).
- Do not prepend indicator variables with a colon (:) in a C Language statement.
- Precede the indicator variable with its associated host variable in an ESQL executable statement.
- Prepend the indicator variable itself with a colon (:) in an ESQL statement.
- Name the indicator variable with a name that is **not** an SQL reserved word.

For more information on indicator variables, see Chapter 8, “Handling NULL Values in ESQL-92,” and Chapter 9, “Handling Errors in ESQL-92.”

3.2.3 Using Indicator Variables with INPUT Host Variables

You can use indicator variables with INPUT host variables. You can assign a NULL value to a column when you INSERT a row into a table or when you UPDATE a table column that allows NULL values.

The code fragment in the following example shows how to declare and use an indicator variable in an INSERT statement. The host variables in this example are: order_no_v, order_date_v, product_v, and qty_v. The indicator variable qty_i is associated with the qty_v host variable. The optional INDICATOR keyword identifies qty_i as an indicator variable.

EXAMPLE

The code fragment illustrates these steps:

1. DECLARE host variables and one indicator variable in the DECLARE SECTION.

2. Set the indicator variable to the value -1 in a C Language section; do not prepend a colon in native C Language statements.

3. INSERT a row into the orders table; set the qty column to NULL; prepend a colon to each host variable and indicator variable in this executable statement.

```
/*
** 1. DECLARE host and indicator variables in the DECLARE SECTION
*/
EXEC SQL BEGIN DECLARE SECTION ;

     long   order_no_v ;
     char   order_date_v[10]
     char   product_v[5] ;
     long   qty_v ;
     long   qty_i ;

EXEC SQL END DECLARE SECTION ;

/*
**    C Language processing here, to assign values for order_no_v,
**    product_v, and to determine qty_v is NULL
*/
   .
   .
   .
/*
** 2. Set indicator variable qty_i for the qty_v host variable;
**    Set to -1 to indicate IS NULL, and there is no value in qty_v
*/

qty_i = -1 ;

/*
** 3. INSERT a row in qty column of orders table with qty_v column NULL
*/

EXEC SQL
     INSERT INTO orders (order_no, product, qty)
     VALUES (:order_no_v, :product_v, :qty_v INDICATOR :qty_i) ;
```

3.2.4 Using Indicator Variables with OUTPUT Host Variables

To use indicator variables to determine if a returned value is a NULL value, you associate an indicator variable with an output host variable. For example, you can use an indicator variable in the INTO clause of a SELECT statement.

These are the general steps for using indicator variables with OUTPUT host variables:

1. DECLARE host variables and an indicator variable in the DECLARE SECTION.
2. Execute the embedded SQL SELECT statement with an INTO clause using an OUTPUT host variable and its associated indicator variable.
3. Examine the indicator variable email_i to determine if the fetched value is NULL.

4. Report the result of the fetch operation.

Table 3–3 describes how you might use indicator variables and INPUT and OUTPUT host variables in the INTO clause of a SELECT statement.

Table 3–3: Host Variables in an OUTPUT Example

Variable Name	Type of Variable	Use in Example
last_name_v	INPUT host variable	C processing determines a value. The SELECT statement WHERE clause uses the value to match last_name column.
email_addr_v	OUTPUT host variable	Target variable for SELECT INTO (OUTPUT) operation. Associated with indicator variable email_i.
email_i	indicator variable	Associated with host variable email_addr_v. C processing evaluates indicator variable for output value, NULL, or truncation.

EXAMPLE

The code fragment in the following example shows how to declare and use indicator variables with the INTO clause of a SELECT statement:

```
/*
**  1. Define host and indicator variables in the DECLARE SECTION
*/
EXEC SQL BEGIN DECLARE SECTION ;
     char last_name_v[20];
     char email_addr_v[20] ;
     long email_i ;
EXEC SQL END DECLARE SECTION ;
/*
**  C Language statements here, to set a value for the INOUT host
**  variable last_name_v. Typically, accept a string from a user
**  interface.
*/
/*
**  2. Execute the SELECT statement with an INTO clause.
*/
EXEC SQL
     SELECT email_addr
     INTO :email_addr_v INDICATOR:email_i
     FROM PERSONNEL
     WHERE last_name = :last_name_v ;
/*
**  Evaluate the SQLCODE and take appropriate program action.
*/
if (SQLCA.SQLCODE == 0)
/*
**  Successful SELECT   3. Evaluate the indicator variable
**                      4. Take appropriate program action
*/
{
     if (email_i == 0)
          printf ("Email address for %s is: %s\n",last_name_v,
                   email_addr_v) ;
     elseif (email_i == -1)
          printf ("No email address on record for %s\n", last_name_v) ;
     else          printf ("Email address for %s too long to process\n",
                   last_name_v) ;
}
else
/*
** SQL_NOT_FOUND and Error Handling Routines here
*/
```

3.2.5 Limitations of the DECLARE SECTION

The ESQL precompiler does not allow you to declare all of the types of variables found in the C Language. These are the operations that are **not valid** in the DECLARE SECTION:

- Referring to names declared in C Language TYPEDEF statements. These names are unknown to the ESQL precompiler, and would result in an error condition.
- Declaring a variable to be a C Language STRUCTURE type.
- Declaring pointers as host variables.

3.3 Types of Executable Statements

There are four categories of ESQL executable statements:

- DML statements

  These statements perform data manipulation operations that can be used to manipulate the data in the database. The DML operations are SELECT, INSERT, UPDATE and DELETE. DML statements are the most frequently used statements in ESQL programs.

- DDL statements

  Use these statements to perform data definitions for a given database. DDL includes statements to CREATE tables or views and to DROP tables or views. Progress SQL-92 provides extensions to the Data Definition Language, including ALTER USER, CREATE SYNONYM, CREATE USER, DROP SYNONYM, and DROP USER.

- DCL statements

  Use these statements to maintain the security of the database. The statements are GRANT and REVOKE.

- Transaction management statements

  Use these statements to manage the start and end of transactions. The statements are COMMIT and ROLLBACK.

3.4 Error Handling in ESQL Programs

This section describes how to use the SQL Communications Area (SQLCA) to test for errors, and how to use the WHENEVER statement to manage exception conditions.

3.4.1 SQL Communications Area (SQLCA)

The SQL engine returns a status code after it processes an SQL executable statement. The status code is returned in the SQL Communications Area (SQLCA). This structure contains information about the status of the execution of the most recently executed SQL statement. The SQLCA provides additional information about the executed statement. The SQLCA includes the following information:

- Warning flags
- Error code
- Diagnostic text
- Number of rows processed for INSERT, UPDATE, and DELETE statements

The SQLCA is a mechanism that allows the application developer to take appropriate program steps, depending on feedback about attempted SQL operations. Your application should examine the SQLCA to determine whether an SQL statement executed successfully. The SQLCA also returns a row count after each INSERT, UPDATE, or DELETE operation.

Two frequently used SQLCA components are:

SQLCODE

> This component holds a status code after the execution of every executable SQL statement. The SQLCODE field is of data type LONG, and its value indicates the success or failure of the statement execution. A zero value returned in SQLCODE indicates successful execution. A negative SQLCODE indicates an error in execution. A positive SQLCODE indicates a successful execution with a status code. Currently, the only positive status code is SQL_NOT_FOUND, which is returned when there are no more rows found in a fetch operation.

SQLWARN

> This component is a CHAR array of size 8, where each array element is set to the warning flag (W) or a blank. SQLWARN[0] is set to W if any of the other flags are set, indicating that a warning level condition occurred during execution.

3.4.2 Using the WHENEVER Statement

An alternative way to handle errors is to use the WHENEVER statement. This statement allows an application to take specific program actions in response to exception conditions. These exceptions include NOT FOUND, SQLERROR, and SQLWARNING. The program actions could be to exit the program, branch to an error handler, or continue execution.

For more information on error handling and a listing of the SQLCA structure, see Chapter 9, "Handling Errors in ESQL-92."

3.5 Transaction Management Statements

When the SQL engine executes an SQL statement, it does so in the context of a transaction. A transaction defines a logical unit of work consisting of a set of database changes that must be executed completely or not at all. The execution of the first SQL executable statement in an ESQL program starts a transaction. All subsequent SQL statements are executed as part of this transaction until you explicitly terminate the transaction or execution terminates.

The primary statements that are used for transaction management are COMMIT and ROLLBACK. Use the COMMIT statement to signal the end of a transaction, and to make permanent the changes made in the database for that transaction. After a COMMIT or ROLLBACK, the first execution of an SQL statement starts a new transaction.

The successful termination of a transaction occurs with the execution of a COMMIT statement. In contrast, the ROLLBACK statement means abnormal termination of a transaction. If a transaction executes the ROLLBACK statement, all changes made to the database in that transaction are canceled.

Good programming practice places the evaluation of the SQLCA in a subroutine, which the program invokes after the execution of an SQL statement or transaction. Fetch operations with SELECT statements require a different subroutine, for evaluation of the SQL_NOT_FOUND condition.

EXAMPLE

This code fragment shows an UPDATE transaction in an ESQL statement. After executing the INSERT, the program logic examines the SQLCA to determine if the operation succeeded or failed. The program issues the appropriate COMMIT or ROLLBACK statement:

```
EXEC SQL
     UPDATE orders
     SET qty = 2000
     WHERE order_no = 1001 ;
/*
** Examine the SQLCODE field in the SQLCA to determine SUCCESS/FAILURE
** Typically this evaluation is in a subroutine.
*/
if (sqlca.sqlcode == 0)
{
     EXEC SQL COMMIT WORK ;
}
else
{
     if (sqlca.sqlcode < 0)
     {          EXEC SQL ROLLBACK WORK ;
          /*
          ** Additional error processing here as needed
          */
     }
}
```

NOTE: For more information on transaction management statements see Chapter 11, "Transaction Management in ESQL-92."

4

Connection Management in ESQL-92

This chapter describes the statements you use for connection management in an ESQL program:

- CONNECT
- SET CONNECTION
- DISCONNECT

4.1 Introduction

Chapter 3, "ESQL-92 Program Structure" describes how to use ESQL statements to access and manipulate data in a database. Before you can gain access to data, you must have a valid connection to the database. You manage this access with *connection management statements.* Use connection management statements to establish a connection to a database, to set an existing connection as current, or to drop an existing connection. These are the connection management statements in SQL-92:

- CONNECT
- SET CONNECTION
- DISCONNECT

The CONNECT statement establishes a connection from an ESQL application to a database. The database you specify in the CONNECT statement becomes the current connection.

NOTE: The server for the database must be running before an ESQL program can connect to it.

The SET CONNECTION statement allows the application to set a particular database connection to the status of current. The DISCONNECT statement terminates the connection between an application and the associated database. Each of these statements is described in a following section.

4.2 CONNECT Statement

The CONNECT statement allows an ESQL application to establish a connection to a database. Different types of database connections are available through an ESQL program. These are:

- CONNECT using a *connection name*
- CONNECT by *DEFAULT*
- CONNECT to a *remote database*

4.2.1 CONNECT Using a Connection Name

A CONNECT statement accepts a *connect_string* and a *connection_name* as arguments to establish a valid connection:

SYNTAX

```
EXEC SQL
CONNECT TO 'connect_string'
  [ AS connection_name ]
  [ USER username ]
  [ USING password ] ;
```

connect_string:

The string that specifies the database to which you are connecting. The *connect_string* must be either string literals enclosed in quotation marks or character-string C Language variables.

SYNTAX

```
DEFAULT
  | db_name
  | progress:T:host_name:port_num:db_name
```

DEFAULT

The SQL engine attempts to connect to the environment-defined database. On both UNIX and Windows-NT platforms the value of the DB_NAME environment variable determines the DEFAULT *connect_string*.

db_name

Name of the database.

progress:T:*host_name*:*port_num*:*db_name*

Connect to a Progress database.

Direct SQL to connect using the TCP/IP protocol for either a local or remote connection.

Specify the name of the system where the database resides. You can specify **localhost** for a local connection.

Number of the communications port where the SQL Server is running. Corresponds to the -S start-up parameter.

Name of the database.

connection_name

The name of the connection to use in CONNECT, DISCONNECT and SET CONNECTION statements. The *connection_name* must be either string literals enclosed in quotation marks or character-string C Language variables.

If a CONNECT statement omits the optional *connection_name*, the SQL engine assigns a *connection_name* which is the same as the database name. Connection names must be unique.

username

User name for authentication of the connection. The SQL engine verifies the *username* against a corresponding *password* before it connects to the database. On both UNIX and Windows-NT platforms, the value of the DH_USER environment variable determines the default *username*. If DH_USER is not set, the value of the USER environment variable determines the default *username*.

password

Password for authentication of the connection. The SQL engine verifies the *password* against a corresponding *username* before it connects to the database.

EXAMPLES

This example establishes a connection to a database using the *connection_name* conn_1:

```
EXEC SQL
     CONNECT TO 'progress:T:localhost:6745:salesdb' AS 'conn_1' ;
```

The *connection_name* must be unique. If you do not specify a *connection_name* the value of the *connect_string* is used as the name of the connection.

For example, consider the above example without the *connection_name* specification. The connection is assigned the name of the value specified in the *connect_string*:

```
EXEC SQL
     CONNECT TO 'progress:T:localhost:6745:salesdb' ;
```

4.2.2 CONNECT by DEFAULT

The CONNECT statement with the DEFAULT keyword establishes a connection to a default database. The default database is the one set in the environment variable DB_NAME.

To connect to the DEFAULT database, use this CONNECT statement:

```
EXEC SQL
     CONNECT TO DEFAULT;
```

The CONNECT statement does not specify a *connection_name*. No connection name is needed for the DEFAULT connection since you can always refer to this connection with the keyword DEFAULT. If an application executes an SQL statement before establishing a connection to a database, the SQL engine attempts to connect to an environment-defined database. If the connection is successful, the database executes the SQL statement.

4.2.3 CONNECT to a Remote Database

The *connect_string* specifies the database type, network protocol, the target host for the database, the port number, and the database name:

```
EXEC SQL
     CONNECT TO 'progress:T:rockwell:6770:salesdb' as 'conn_2' ;
```

4.3 SET CONNECTION Statement

Use the SET CONNECTION statement to switch the application from one established connection to another. This resumes the connection associated with the specified *connection_name*, restoring the context of that database connection to the state it was in when suspended.

SYNTAX

```
EXEC SQL
  SET CONNECTION { 'connection_name' | DEFAULT } ;
```

EXAMPLES

The first example shows how to establish a database as the current database. The SET CONNECTION command sets the database associated with the connection named 'conn_1' to the status of current database. The connection named 'conn_1' must be associated with an established connection:

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

Use this statement to **set current** the database associated with the DEFAULT connection. This statement changes the state of the earlier 'conn_1' connection from current to suspended:

```
EXEC SQL
     SET CONNECTION DEFAULT ;
```

4.4 DISCONNECT Statement

The DISCONNECT statement terminates the connection between an application and the database to which it is connected:

SYNTAX

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

connection_name

Disconnects the named database. If there is also an established connection to the DEFAULT database, the connection to the DEFAULT database becomes the current connection. If there is no DEFAULT database, there is no current connection after the SQL engine processes the DISCONNECT.

CURRENT

Disconnects the CURRENT database. f there is also an established connection to the DEFAULT database, the connection to the DEFAULT database becomes the current connection. If there is no DEFAULT database, there is no current connection after the SQL engine processes the DISCONNECT.

ALL

Disconnects all established connections. After you issue DISCONNECT ALL, there is no CURRENT connection.

DEFAULT

Terminates the connection to the DEFAULT database. If this connection is the current connection, there is no current connection after this DISCONNECT statement is executed.

EXAMPLES

The first example illustrates CONNECT TO AS *'connection_name'* and DISCONNECT *'connection_name'*:

```
EXEC SQL
     CONNECT TO 'progress:T:localhost:6745:salesdb' AS 'conn_1' ;
/*
**  C Language and embedded SQL-92 application processing against the
**  database in the connect_string
*/
   .
   .
   .
EXEC SQL
     DISCONNECT 'conn_1' ;
```

This example illustrates CONNECT TO DEFAULT and DISCONNECT DEFAULT:

```
EXEC SQL
    CONNECT TO DEFAULT ;
/*
**  C Language and embedded SQL-92 application processing against the
**  database in the connect_string
*/
   .
   .
   .
EXEC SQL
    DISCONNECT DEFAULT ;
```

When you specify the ALL option, all established connections are disconnected. After you issue DISCONNECT ALL there is no current connection. For example:

```
EXEC SQL
     DISCONNECT ALL;
```

The following example illustrates these steps:

- CONNECT TO '*connect_string*' AS '*connection_name*' establishes a connection to the database in the *connect_string;* the connection has the name 'conn_1'.
- CONNECT TO DEFAULT establishes a connection to the DEFAULT database and sets this connection current.
- DISCONNECT DEFAULT disconnects the connection to the DEFAULT database.
- SET CONNECTION 'conn_1' sets the 'conn_1' connection current.
- DISCONNECT CURRENT disconnects the 'conn_1' connection.

```
EXEC SQL
     CONNECT TO 'progress:T:localhost:6745:salesdb' AS 'conn_1' ;
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** Application processing against the DEFAULT database
*/
   .
   .
   .
EXEC SQL
     DISCONNECT DEFAULT ;
EXEC SQL
     SET CONNECTION 'conn_1' ;
/*
** Application processing against the database in the connect_string
*/
   .
   .
   .
EXEC SQL
     DISCONNECT CURRENT ;
```

5

SQL-92 Data Definition Statements

This chapter introduces the *Data Definition Language* (DDL) statements that you can use in an ESQL program. The statements discussed here include statements to CREATE and DROP tables, views, and indexes. This chapter also provides a description of the *integrity constraints* that are available in DDL syntax. The last section discusses DDL statements in long-running transactions.

See Chapter 2, “SQL-92 Statements,” in the *Progress SQL-92 Guide and Reference* for complete syntax of supported DDL statements.

5.1 CREATE TABLE and DROP TABLE Statements

This section discusses creating and dropping tables in an ESQL program.

5.1.1 CREATE TABLE

The CREATE TABLE statement allows you to create a new table in an existing database by defining its column names and column data types. You specify the table name for the table you are creating. Optionally, a CREATE TABLE statement can include table and column constraints.

EXAMPLE

The following code fragment illustrates a CREATE TABLE statement in an ESQL program. The cust_no column has the column constraint NOT NULL, which indicates that no row in the customer table is to have a NULL value in the cust_no column:

(1 of 2)

```
/*
** CONNECT TO the DEFAULT database
*/
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** CREATE a table with the table name customer.
*/
EXEC SQL CREATE TABLE customer
     (
     cust_no INTEGER NOT NULL,
     last_name CHAR (30),
     street CHAR (30),
     city CHAR (20),
     state CHAR (2)
     ) ;
```

(2 of 2)

```
if (sqlca.sqlcode < 0)
{
     printf ("Create table statement failed (%ld : %s) \n",
               sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1) ;
}
/*
**  Commit the CREATE TABLE operation.
*/
EXEC SQL
     COMMIT WORK ;
printf ("Created customer table. \n");
/*
**  DISCONNECT from the DEFAULT database
*/
EXEC SQL
     DISCONNECT DEFAULT ;
```

The CREATE TABLE statement also allows you to specify the DEFAULT clause along with a column definition. The DEFAULT clause identifies the default value to be used for a column, if you do not supply a value for that column in an INSERT statement.

EXAMPLE

The following CREATE TABLE statement shows how to use the DEFAULT clause. This example sets a default value of 10 for the deptno column:

```
EXEC SQL
CREATE TABLE employee
     (
     empno   INTEGER NOT NULL,
     deptno  INTEGER DEFAULT 10
     ) ;
```

5.1.2 DROP TABLE

The DROP TABLE statement deletes all data and indexes for a table and erases its entry in the system catalog.

EXAMPLE

The following code fragment shows a DROP TABLE statement to drop the tmp_customer table:

```
/*
** CONNECT TO the DEFAULT database
*/
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** Drop the tmp_customer table from the database.
*/
EXEC SQL
     DROP TABLE tmp_customer ;

if (sqlca.sqlcode < 0)
{
     printf ("DROP TABLE statement failed (%ld : %s) \n",
          sqlca.sqlcode, sqlca.sqlerrm) ;
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1) ;
}
/*
** Successful DROP; COMMIT the DROP operation.
*/
EXEC SQL
     COMMIT WORK ;
printf ("Dropped tmp_customer table. \n");

/*
** DISCONNECT from the DEFAULT database
*/
EXEC SQL
     DISCONNECT DEFAULT ;
```

5.2 CREATE INDEX and DROP INDEX Statements

This section illustrates how to CREATE an index and DROP an index using ESQL.

5.2.1 CREATE INDEX

The CREATE INDEX statement creates an index on one or more columns of a table. The purpose of an index is to improve the retrieval time for rows in a table. You can specify an index in ascending order (ASC) or descending order (DESC).

EXAMPLE

The index in this CREATE INDEX example is specified on the single column cust_no, and is of ascending order on the value of the column:

```
/*
** CONNECT to the DEFAULT database.
*/
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** Create the idx_cust.index.
*/
EXEC SQL
     CREATE INDEX idx_cust ON customer (cust_no ASC) ;
if (sqlca.sqlcode < 0)
{
     printf ("CREATE INDEX idx_cust statement failed (%ld : %s) \n",
          sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}
/*
** COMMIT the CREATE INDEX operation.
*/
EXEC SQL COMMIT WORK ;
printf ("Index idx_cust created. \n");
/*
** DISCONNECT from the DEFAULT database
*/
EXEC SQL
     DISCONNECT DEFAULT ;
```

5.2.2 DROP INDEX

Use the DROP INDEX statement to drop an index on a table.

EXAMPLE

The following code fragment shows a DROP INDEX statement, dropping the idx_cust index on the customer table:

```
/*
** CONNECT to the DEFAULT database.
*/
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** DROP the idx_cust index.
*/
EXEC SQL
     DROP INDEX idx_cust ON customer;
if (sqlca.sqlcode < 0)
{
     printf ("DROP INDEX statement failed (%ld : %s) \n",
             sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}
/*
** Successful drop; commit the drop index operation.
*/
EXEC SQL COMMIT WORK ;
printf ("DROP INDEX idx_cust successful.\n");
/*
** Disconnect from the default database.
*/
EXEC SQL
     DISCONNECT DEFAULT ;
```

5.3 CREATE VIEW and DROP VIEW Statements

This section discusses how to CREATE a view and DROP a view in an ESQL program.

5.3.1 CREATE VIEW

Use the CREATE VIEW statement to create a view on existing tables or views. You specify the name for the VIEW.

EXAMPLE

The following code fragment shows how to use a CREATE VIEW statement in an ESQL program. This CREATE VIEW statement creates the ne_customer view of the customer table:

```
/*
** CONNECT to the DEFAULT database.
*/
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** Create the ne_customer view.
*/
EXEC SQL
     CREATE VIEW ne_customer AS
     SELECT cust_no, last_name, street, city, state
     FROM customer
     WHERE state in ('NH', 'MA', 'ME', 'CT', 'RI', 'VT') ;
if (sqlca.sqlcode < 0)
{
     printf ("CREATE VIEW statement failed (%ld : %s) \n",
             sqlca.sqlcode, sqlca.sqlerrm) ;
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1) ;
}
/*
** Successful create view; commit the operation.
*/
EXEC SQL
     COMMIT WORK ;
printf ("CREATE VIEW ne_customer successful. \n");
/*
** Disconnect from the default database.
*/
EXEC SQL
     DISCONNECT DEFAULT ;
```

5.3.2 DROP VIEW

The DROP VIEW statement deletes an existing view from the database.

EXAMPLE

The following example drops the ne_customer view:

```
/*
** CONNECT to the DEFAULT database.
*/
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** Drop the ne_customer view.
*/
EXEC SQL
     DROP VIEW ne_customer ;
if (sqlca.sqlcode < 0)
{\n
     printf ("DROP VIEW statement failed (%ld : %s) ",
          sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}
/*
** Successful drop; commit the operation.
*/
EXEC SQL
     COMMIT WORK ;
printf ("DROP VIEW ne_customer successful. \n") ;
/*
** Disconnect from the default database
*/
EXEC SQL
     DISCONNECT DEFAULT ;
```

5.4 Integrity Constraints

Integrity constraints are application rules that the database enforces. You define integrity constraints on base tables to ensure data integrity in a database. An integrity constraint can specify unique values for a column, validate values of a column, or provide referential integrity.

Referential integrity ensures that the relationships among different rows of the same or different tables are valid. For example, you might use referential integrity to ensure that a master row is inserted before a detail row, and a detail row is deleted before a master row.

5.4.1 Need for Integrity Constraints

Integrity constraints are necessary because data in a database must be valid and consistent at all times. Data might be inconsistent because of entry errors, duplicate entries of rows, or other violations.

To illustrate, consider a table containing employee information where the employee numbers must be unique. To ensure this, specify a UNIQUE constraint on the column that contains the employee number (emp_no):

```
EXEC SQL
     CREATE TABLE employee_info (
          emp_no      INTEGER NOT NULL UNIQUE,
          first_name  VARCHAR(20) NOT NULL,
          last_name   VARCHAR(20) NOT NULL,
          title       VARCHAR(20)
     ) ;
```

See the "Candidate Keys" section for information on the UNIQUE keyword.

Similarly, for each order entry made in an orders table, you might want to ensure that the associated item entry is present in another table by specifying a referential integrity constraint on a column in the orders table. See the examples in the "Referential Constraints" section later in this chapter.

5.4.2 Types of Integrity Constraints

These are the types of integrity constraints:

- Check constraint
- Primary key specification
- Candidate key specification
- Referential constraint

Optionally, you can name an integrity constraint, and refer to that name in other SQL-92 statements. The database assigns a constraint name if you do not specify one. Typically, application-defined constraint names are easier to manage. Database-defined names can be long, unwieldy, and might require look-up in a system catalog.

EXAMPLE

The following code fragment shows the specification of a table constraint named prim_constr on the supplier_item table. You specify a constraint name with the CONSTRAINT keyword:

```
EXEC SQL
     CREATE TABLE supplier_item (
          supp_no  INTEGER NOT NULL,
          item_no  INTEGER NOT NULL,
          qty      INTEGER NOT NULL DEFAULT 0
          CONSTRAINT prim_constr
          PRIMARY KEY (supp_no, item_no)
     ) ;
```

5.4.3 Check Constraints

The values you enter for a row must be valid so that the data in the database is consistent. For example, the city names you enter into the supplier table must correspond to one of the cities where the suppliers are located. The database checks to ensure that each value corresponds to one of the valid city names. You achieve these validations by specifying check constraints during the definition of the table schema. Use check constraints when you want to restrict a column to a set of valid values.

EXAMPLE

The following code fragment shows how to specify a check constraint on the supplier table. In this example, the city column is defined with a check constraint to verify that values for city are in the set of NEWYORK, BOSTON, DALLAS, or MANCHESTER. This CREATE statement does not use the CONSTRAINT keyword in the table definition. The system assigns a constraint name:

```
EXEC SQL
     CREATE TABLE supplier (
          supp_no  INTEGER NOT NULL,
          last_name  CHAR (30),
          status SMALLINT,
          city  CHAR (20) CHECK (
          supplier.city IN ('NEWYORK', 'BOSTON', 'DALLAS', 'MANCHESTER'))
     ) ;
```

A check constraint on a table specifies a condition on the column values of a row in that table. Whenever you issue an INSERT or UPDATE operation against a table containing check constraints, the database validates the column values. The INSERT or UPDATE operation is completed only after successful validation.

You can specify a check constraint either at the column level or at the table level. The following sections discuss column level and table level check constraints.

Column Level Check Constraint

In an application, you might decide to check a particular column for valid data whenever you attempt to INSERT or UPDATE values for that column. For example, you design your database to disallow suppliers from the city of BadApple. Use a column level check constraint for this type of validation.

EXAMPLE

In the next example there is a column level check constraint on the city column of the supplier table; this check constraint affects the city column only. When you issue an INSERT or UPDATE operation against the supplier table involving the city column, the SQL engine validates the column value, ensuring that the column does not contain the value BadApple. If the INSERT or UPDATE statement violates the check condition, the database returns a constraint violation error:

```
EXEC SQL
CREATE TABLE supplier (
     supp_no  INTEGER NOT NULL,
     last_name  CHAR (30),
     status SMALLINT,
     city  CHAR (20) CHECK (
     supplier.city <> 'BadApple')
     ) ;
```

For example, the following INSERT statement results in an error, and the corresponding row is not inserted into the table:

```
/*
** This INSERT statement FAILS with a constraint violation.
*/
EXEC SQL
     INSERT INTO supplier VALUES (1001, 'Worm', 20, 'BadApple') ;
```

Table Level Check Constraint

Your application might be required to enforce rules on multiple columns. To specify a constraint on more than one column of a table you define the constraint at the table level. For example, you might want to enforce a validation check on both the status and the city columns in the supplier table.

EXAMPLE

In this example, the table level check constraint verifies that when the city **is** CHICAGO, the status must be 20, otherwise the operation returns a table level check constraint violation:

```
EXEC SQL
     CREATE TABLE supplier (
     supp_no    INTEGER NOT NULL,
     last_name       CHAR (30),
     status     SMALLINT CHECK (
          supplier.status BETWEEN 1 AND 100 ),
     city       CHAR (20)
     CHECK (
          supplier.city IN ('NEWYORK', 'BOSTON', 'CHICAGO', 'MANCHESTER' )
            ),
     CHECK (supplier.city <> 'CHICAGO' OR supplier.status = 20)
           )
     ) ;
```

Since the check constraint specification involves more than one column, you must specify it at the table level. If an INSERT or UPDATE statement violates the check condition the database returns an error. The following example shows an INSERT statement for the supplier table created in the previous example. This INSERT operation results in a check constraint violation:

```
/*
** This INSERT violates a CHECK CONSTRAINT.
*/ EXEC SQL
     INSERT INTO supplier VALUES (1001, 'John', 40, 'CHICAGO') ;
```

5.4.4 Primary Keys

Your application might require that a database table contain one or more columns to identify a row uniquely. For example, in the supplier table the supp_no column value must be unique. Every row of the table is uniquely identified by this column value. A single column or a group of columns can be the principal unique identifier of the table. The principal unique identifier is called a *primary key*. A table can contain only one primary key constraint.

If you supply a duplicate value for a primary key column in an INSERT operation, the operation returns an error.

Column Level Primary Key Constraint

You can design your database table so that there is only one column that distinguishes a given row from other rows. In this case, a single column is the unique identifier of the table. For example, the supp_no column is a *column level primary key* for the supplier table. Column level primary key constraints are defined in the column definitions of a table.

EXAMPLE

In this example, the supp_no column is a unique identifier in the supplier table, and the key consists of only one column. This example shows how to create a column level primary key on the supplier table:

```
EXEC SQL
     CREATE TABLE supplier
          (
          supp_no  INTEGER NOT NULL PRIMARY KEY,
          last_name  CHAR (30),
          status  SMALLINT,
          city  CHAR (20)
          ) ;
```

Table Level Primary Key Constraint

You can design your database table to require more than one column to identify a row as unique. The content of your data might require this. You can use a combination of columns to ensure that each row is unique.

EXAMPLE

For example, the columns supp_no and item_no uniquely identify a row in the supplier_item table. You must specify a combination of columns that form a primary key at the table level. The following example shows a *table level primary key* specification:

```
EXEC SQL
     CREATE TABLE supplier_item
          (
          supp_no  INTEGER NOT NULL,
          item_no  INTEGER NOT NULL,
          qty  INTEGER NOT NULL DEFAULT 0
          CONSTRAINT prim_constr
          PRIMARY KEY (supp_no, item_no)
          ) ;
```

Since multiple columns (supp_no, item_no) are in the primary key, you must specify the constraint at the table level. In this example, the constraint named prim_constr is the primary key of the supplier_item table.

5.4.5 Candidate Keys

If you design your table to require that a column or combination of columns define a row as unique, you define the column or columns with a *candidate key* constraint.

EXAMPLE

In the next example, the employee number (empno) in the employee table uniquely identifies the row, and is the primary key for the employee table. There is another column that contains the social security number for the employee. Since the values for this column must be distinct for each row in the table, you should also define the ss_no column with a candidate key constraint:

```
EXEC SQL
     CREATE TABLE employee (
          empno  INTEGER NOT NULL PRIMARY KEY,
          ss_no  INTEGER NOT NULL UNIQUE,
          ename  CHAR (19),
          sal  NUMERIC (10, 2),
          deptno  INTEGER NOT NULL
     ) ;
```

You declare a column as a candidate key by using the keyword UNIQUE. Precede the UNIQUE keyword with the NOT NULL specification for that column. Like a primary key, a candidate key also uniquely identifies a row in a table. Note that a table can have only one primary key, but can have any number of candidate keys.

If you supply a duplicate value for a candidate key in an INSERT or UPDATE operation, the operation returns an error.

Column Level Candidate Key Constraint

You can use a *column level candidate key* constraint to ensure that a single column is unique in every row. For example, the ss_no column in the employee table must be unique. A column level candidate key constraint involves a single column.

EXAMPLE

In this example, the ss_no column is defined as a candidate key in the employee table by specifying the UNIQUE keyword in the column definition:

```
EXEC SQL
     CREATE TABLE employee
     (
          empno  INTEGER NOT NULL PRIMARY KEY,
          ss_no  INTEGER NOT NULL UNIQUE,
          ename  CHAR (19),
          sal  NUMERIC (10, 2),
          deptno  INTEGER NOT NULL
     ) ;
```

Table Level Candidate Key Constraint

If your application requires unique values for a combination of columns, define a *table level candidate key constraint*.

EXAMPLE

For example, in the order_item table the columns order_no and item_no **together** form a unique key. The **combination** of the order_no and item_no columns is a unique identifier in the order_item table, as shown in the following example:

```
EXEC SQL
     CREATE TABLE order_item (
          order_no INTEGER NOT NULL,
          item_no  INTEGER NOT NULL,
          qty  INTEGER
          UNIQUE (order_no, item_no)
     ) ;
```

In the following example, the first two insert statements succeed. Each of these statements inserts a row in the order_item table, and each specifies the value 322 for order_no. There is no column level candidate key constraint on the order_no column. The combination of values for the order_no and item_no columns is unique. The third insert statement fails with a table level candidate key constraint violation. The final insert statement fails with a violation of the not null constraint:

```
/*
** 1. This statement succeeds, inserting one row.
*/
EXEC SQL
     INSERT INTO order_item
          VALUES ( 322, 55, 288 ) ;
/*
** 2. This statement supplies the same value, 322, for order_no.
**    The insert succeeds, since the combination of the values for
**    order_no and item_no is unique.
*/
EXEC SQL
     INSERT INTO order_item
          VALUES ( 322, 56, 288 ) ;
/*
** 3. This statement fails with a table level candidate key constraint
**    violation. The database does not allow two rows with the same
**    order_no AND item_no.
*/
EXEC SQL
     INSERT INTO order_item
          VALUES ( 322, 55, 144 ) ;
/*
** 4. This statement violates the NOT NULL constraint.
**    The operation returns this error:
**    "Null value supplied for a mandatory (not null) column."
*/
EXEC SQL
     INSERT INTO order_item (item_no, qty)
          VALUES ( 56, 288) ;
```

5.4.6 Referential Constraints

Sometimes specific columns in tables in two different locations in a database must be identical. The values for each row in a column or group of columns taken together must be identical to a set of columns in another table. You define the requirement that a matching row must exist in the referenced table with a *referential constraint*. For example, the employee number of the employee table might be in the given range, but the employee number might no longer exist in the master employee table because the employee has resigned. Use referential constraints to enforce this kind of limit.

EXAMPLE

In the next example, the value in the item_no column of the supplier_item table depends on the value in the item_no column of the item table. The item_no column of the supplier_item table references the item_no column of the item table. The item_no column is a *foreign key* in the supplier_item table:

```
EXEC SQL
     CREATE TABLE supplier_item
          (
         suppl_no  INTEGER NOT NULL PRIMARY KEY,
         item_no   INTEGER REFERENCES item (item_no),
         quantity  INTEGER
          ) ;
```

A foreign key is a column or combination of columns that references a primary key or a candidate key of some table. The foreign key value is either NULL or exists as the primary key value. The table that contains the foreign key is called the referencing table. The table that contains the primary or the candidate key is called the referenced table. You can specify a referential constraint at the column level or at the table level.

During INSERT or UPDATE operations on a table containing a foreign key, the database checks to determine if the foreign key value matches a corresponding primary key value. If it does not match, the operation returns an error.

During UPDATE or DELETE operations on a table containing a primary or candidate key, if the values to be deleted or updated match the foreign key of the referencing table, the operation returns an error. A value corresponding to a primary or candidate key cannot be updated or deleted if there are references to it.

When you want to drop a table containing a primary or candidate key, the database checks to see if the table has any references to it. If there are tables containing foreign keys that reference the primary or candidate keys of the table you want to drop, the operation returns an error.

Column Level Foreign Key Constraint

If a foreign key constraint specification involves only one column, you specify a *column level foreign key constraint*.

EXAMPLE

In the next example, item_no is the foreign key referencing the item table, and the foreign key is specified at the column level:

```
EXEC SQL
     CREATE TABLE supplier_item
           (
          supp_no  INTEGER NOT NULL PRIMARY KEY,
          item_no  INTEGER NOT NULL REFERENCES item,
          qty  INTEGER
          ) ;
```

If a foreign key references a candidate key, you must name the referenced column in a column list. If a foreign key references a primary key, the column list is optional.

EXAMPLE

The next example illustrates both conditions. In the example, invoice.item_no references the primary key of the item table. The invoice.partnum column references parts.part_no. Since parts.part_no is a primary key, the parts (part_no) column list reference in invoice.part_no is optional:

```
EXEC SQL
     CREATE TABLE invoice (
          inv_no  INTEGER NOT NULL PRIMARY KEY,
          item_no  INTEGER REFERENCES item,
          part_no  CHAR(3) NOT NULL
               REFERENCES parts (part_no),
          qty   INTEGER NOT NULL
     ) ;
```

Table Level Foreign Key Constraint

If a foreign key constraint specification involves more than one column, you must specify the constraint at the table level.

EXAMPLE

In this example, the foreign key (empno, projno) of the hours_worked table references the primary or candidate key (empno, projno) of the assignments table:

```
EXEC SQL
     CREATE TABLE hours_worked
           (
          empno  INTEGER NOT NULL,
          projno INTEGER NOT NULL,
          date  DATE,
          hours  TIME
          FOREIGN KEY (empno, projno)
               REFERENCES assignments (empno, projno)
          ) ;
```

5.4.7 Handling Cycles in Referential Integrity

A *cycle* is a specific relationship between base tables. A cycle exists when there is a list of two or more base tables, and the first table has a foreign key that references the second table, the second table has a foreign key that references the third table and so on, and the last table has a foreign key that references the first table.

CAUTION: Release 9.0B of Progress SQL-92 does not supply a mechanism for dropping a cycle. You can create a deadlock situation if you define tables with referential integrity constraints in a cycle. Create these in a test database only. The only mechanism for dropping these tables is to delete the database.

EXAMPLE

In the next example, the parts.distrib_no column references the primary key of the distributor table, and the distributor.part_no column references the primary key of the parts table. Each of the tables references the other, forming a cycle:

```
EXEC SQL
     CREATE TABLE parts
          (
          part_no  INTEGER NOT NULL PRIMARY KEY,
          part_name  CHAR (19),
          distrib_no INTEGER REFERENCES distributor
          ) ;
EXEC SQL
     CREATE TABLE distributor
          (
          distrib_no  INTEGER NOT NULL PRIMARY KEY,
          distrib_name  CHAR (19),
          address   CHAR (30),
          phone_no  CHAR (10),
          part_no   INTEGER REFERENCES parts
          ) ;
```

EXAMPLE

A special case of the cycle in referential integrity occurs when a foreign key of a table references the primary key of the same table. The following example shows this single-table cycle:

```
EXEC SQL
     CREATE TABLE employee
          (
          empno INTEGER NOT NULL PRIMARY KEY,
          ename CHAR (30) NOT NULL,
          deptno  INTEGER NOT NULL,
          mgr_code INTEGER REFERENCES employee(empno)
          ) ;
```

Creating Tables in Cycles

Follow these general steps to create a table cycle:

1 ♦ Create the first table with a reference to a table that is not yet created. Although the CREATE TABLE succeeds, it is marked incomplete. The INSERT, UPDATE, SELECT, and DELETE operations are not allowed on an incomplete table.

2 ♦ Create the referenced table with a primary or candidate key. The definition of the referencing table, the first table, becomes complete. If this second table also contains a foreign key that references a table that is not yet created, this second table is also marked incomplete. This process continues until you create the last table.

Inserting Rows in a Cycle

Follow these general steps to insert rows into tables that form a cycle:

1 ♦ Insert rows into one of the tables that forms the cycle, with NULL values in the foreign key columns. If the foreign key is NULL, the database does not check for a match between the foreign key and the corresponding primary key. The insert succeeds. This is the referencing table.

2 ♦ Update or insert the values in the primary keys of the second table, the referenced table.

3 ♦ Update the foreign key values of the previous table, the referencing table.

EXAMPLE

The next example shows how to insert or update values into the employee table. This table forms a single-table cycle. First insert NULL into the mgr_code column. After you insert rows, update the values of the mgr_code column:

```
EXEC SQL
     CREATE TABLE employee (
          empno INTEGER NOT NULL PRIMARY KEY,
          ename CHAR (30) NOT NULL,
          deptno  INTEGER NOT NULL,
          mgr_code INTEGER REFERENCES employee (empno)
     ) ;
EXEC SQL
     INSERT INTO employee VALUES (100, 'JOHN', 10, NULL) ;
EXEC SQL
     INSERT INTO employee VALUES (500, 'MARY', 30, NULL) ;
EXEC SQL
     INSERT INTO employee VALUES (101, 'ANITA', 10, NULL) ;
EXEC SQL
     INSERT INTO employee VALUES (501, 'ROBERT', 30, NULL) ;
EXEC SQL
     UPDATE employee set mgr_code = 101 where empno = 100 ;
EXEC SQL
     UPDATE employee set mgr_code = 501 where empno = 500 ;
/*
** Anita is John's manager.
** John's employee row references Anita's employee row.
** Robert is Mary's manager.
** Mary's employee row references Robert's employee row.
** The mgr_code is still NULL in Anita's row and in Robert's row.
** To set the mgr_code in Anita's row and Robert's row:
**      1. Insert rows for Anita's manager and Robert's manager
**      2. Update Anita's row and Robert's row
*/
```

5.5 DDL Statements in Long-running Transactions

In an ESQL program, DDL statements have the same transaction semantics as other SQL statements. You must explicitly COMMIT or ROLLBACK DDL operations in an ESQL application.

You can execute DDL statements as part of a long-running transaction. To avoid locking database resources such as tables and views for a long period, Progress Software recommends that you commit the DDL operations. This also improves the concurrency of database operations.

6

SQL-92 Data Manipulation Statements

This chapter describes using *Data Manipulation Language* (DML) statements in ESQL programs. It discusses the types of DML statements, including SELECT, INSERT, DELETE, and UPDATE. This chapter also discusses INPUT host variables that can be used in DML statements.

See Chapter 3, "ESQL-92 Program Structure," for information on indicator variables and more information on how to use host variables.

See the "SQL-92 Statements" chapter in the *Progress SQL-92 Guide and Reference* for complete syntax of supported DML statements.

6.1 Using DML Statements

Data Manipulation Language (DML) statements are the most frequently used statements in an ESQL program. The SELECT statement, also called a query statement, is one type of DML statement. You use a SELECT statement to retrieve data from a database. See Chapter 7, "Query Statements," for information on how to use SELECT statements.

You use DML statements to change data in a database in one of three ways:

- INSERT — Add one or more rows to a table.
- DELETE — Delete one or more rows from a table.
- UPDATE — Modify the data in one or more rows of a table.

Use DML statements to modify only one table at a time. You can reference host variables in a DML statement. The INSERT, UPDATE, and DELETE statements are discussed in more detail in following sections.

6.1.1 INSERT Rows into a Table

An INSERT statement adds one or more rows to an existing table.

EXAMPLE

The following INSERT statement adds new customer 'Nyquist' to the customer table:

```
EXEC SQL
     INSERT INTO customer
          (cust_no, last_name, street, city, state)
     VALUES
          (1006, 'Nyquist', '1 Perpetual Street', 'St. Cloud', 'MN') ;
```

INSERT a Single Row into a Table

When you use an INSERT statement in an ESQL program, you can use host variables to supply the values. The columns of the customer table listed in the INSERT statement are referred to as the column list. The host variables specified in the VALUES clause of the INSERT statement are referred to as the value list.

EXAMPLE

In the next example, the values for the host variables are determined with program logic, and inserted into the customer table using the INSERT statement:

```
EXEC SQL BEGIN DECLARE SECTION ;
     long cust_no_v ;
     char name_v [20] ;
     char street_v [40] ;
     char city_v [10] ;
     char state_v [2] ;

EXEC SQL END DECLARE SECTION ;

/* Connect to the default database */
EXEC SQL CONNECT TO DEFAULT ;

/* Assign values for input host variables */
cust_no_v = 1006 ;
strcpy (name_v, "Nyquist") ;
strcpy (street_v, "1 Perpetual Street") ;
strcpy (city_v, "St. Cloud") ;
strcpy (state_v, "MN") ;

EXEC SQL
     INSERT INTO customer
          (cust_no, last_name, street, city, state)
     VALUES
          (:cust_no_v, :name_v, :street_v, :city_v, :state_v) ;

if (sqlca.sqlcode < 0)
{
     printf ("Insert statement failed (%ld : %s). \n",
          sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}

/* Success; commit the insert operation */
EXEC SQL COMMIT WORK ;
printf ("Inserted one row \n");

/*
** Disconnect from the default database
*/
EXEC SQL DISCONNECT DEFAULT ;
```

INSERT Multiple Rows into a Table

To INSERT more than one row, execute an INSERT statement with a sub-query.

EXAMPLE

The following code fragment shows how to INSERT multiple rows from the customer table into the mn_customer table:

```
EXEC SQL
     CREATE TABLE mn_customer (
      cust_no  INTEGER,
      last_name   CHAR(20),
      street  CHAR(40),
      city   CHAR(15),
      state  CHAR(2)
     );



EXEC SQL
     INSERT INTO mn_customer
          (cust_no, last_name, street, city, state)
          SELECT cust_no, last_name, street, city, state
          FROM customer
          WHERE state = 'MN' ;
```

The mn_customer table must already exist before you can INSERT rows. The SELECT query expression option allows you to INSERT multiple rows at a time. The query expression must successfully return values for **all** the columns in the INSERT statement for the operation to succeed.

6.1.2 DELETE Rows from a Table

A DELETE statement deletes one or more rows from an existing table, depending on the selection criteria in the WHERE clause.

CAUTION: If you do not specify a WHERE clause, the DELETE statement deletes all rows from the table.

EXAMPLE

The following code fragment shows how to use a DELETE statement to delete a row from the customer table. The program uses the host variable cust_no_v to match a row with cust_no 1005, and deletes the row from the table. If the cust_no column is a primary key or is a candidate key, one row is deleted. If there are multiple rows with cust_no = 1005, multiple rows are deleted:

```
/* CONNECT to the DEFAULT database */
EXEC SQL CONNECT TO DEFAULT ;

/* Get value for cust_no_v */
cust_no_v = 1005 ;

EXEC SQL
     DELETE
     FROM customer
     WHERE cust_no = :cust_no_v;

if (sqlca.sqlcode < 0)
{
     printf ("DELETE statement failed (%ld : %s)\n",
               sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}
/* Successful; commit the delete operation.*/
EXEC SQL COMMIT WORK ;

printf ("Deleted one row \n\n");

/* DISCONNECT from the DEFAULT database */
EXEC SQL DISCONNECT DEFAULT ;
```

A DELETE operates on more than one row if the WHERE clause matches multiple rows.

EXAMPLE

The following example deletes any rows from the orders table where the value in the order_date column is less than 2/2/1999:

```
EXEC SQL
     DELETE
     FROM orders
     WHERE order_date < TO_DATE ('02/02/1999') ;
```

6.1.3 UPDATE Rows in a Table

An UPDATE statement modifies data in one or more rows of a table.

EXAMPLE

The following UPDATE statement updates the phone number of a row in the customer table:

```
/*
** Connect to the default database.
*/
EXEC SQL CONNECT TO DEFAULT ;

cust_no_v = 1004 ;

EXEC SQL
     UPDATE customer
     SET  phone = '(203)555-2703'
     WHERE cust_no = :cust_no_v ;

if (sqlca.sqlcode < 0)
{
     printf ("Update statement failed (%ld : %s)\n",
               sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}
/*
** Successful; COMMIT the UPDATE operation.
*/
EXEC SQL COMMIT WORK ;

printf ("Update Successful\n\n");

/* Disconnect from the database */
EXEC SQL DISCONNECT DEFAULT ;
```

EXAMPLE

The following UPDATE statement gives a 10 percent increase in salary to all employees of department 12. Note that multiple rows are updated with this UPDATE statement:

```
EXEC SQL
    UPDATE employee
    SET sal = sal * 1.1
    WHERE deptno = 12 ;
```

6.2 INPUT Host Variables in DML Statements

The host variables that you use for input in an SQL statement are referred to as *INPUT host variables*. These variables must be declared in the DECLARE SECTION before you can use them in ESQL executable statements. You can use them in ESQL statements wherever a constant is valid. INSERT, UPDATE, and DELETE DML statements reference INPUT host variables.

EXAMPLE

The following example shows input host variable references in DML statements:

```
EXEC SQL
   INSERT INTO customer
       (cust_no, last_name, street, city, state)
   VALUES       (:cust_no_v, :name_v, :street_v, :city_v, :state_v) ;


EXEC SQL
    UPDATE customer
    SET  phone = '(203) 555-2793'
    WHERE cust_no = :cust_no_v ;
```

NOTE: Prepend a colon (:) to any host variables you use in ESQL executable statements.

7

Query Statements

This chapter describes SQL queries and how to use them in ESQL programs. It defines support for the SELECT statement. Specifically, this chapter includes these topics:

- Elements of a query
- Query statements with host variables
- Queries returning a single row
- Queries returning multiple rows
- Using cursors
- Array fetches

7.1 Elements of a Query

ESQL supports SELECT statements for performing queries. A SELECT statement can return one or more rows.

EXAMPLE

The following code fragment shows a SELECT statement that retrieves the values of the last_name and city columns for a specific customer in the customer table:

```
EXEC SQL BEGIN DECLARE SECTION ;
     CHAR  name_v [30] ;
     CHAR  city_v [20] ;
     LONG  cust_no_v ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
/*
** Application processing here, to assign values to host variables.
*/

EXEC SQL
     SELECT last_name, city
     INTO :name_v, :city_v
     FROM customer
     WHERE cust_no = :cust_no_v;
```

7.1.1 SELECT Statement Clauses

The example in the "Elements of a Query" section contains the four clauses of the SELECT statement:

- SELECT clause
- INTO clause
- FROM clause
- WHERE clause

The column names specified in the SELECT clause are referred to as the SELECT list or projection. In this example, the SELECT list includes the last_name and city columns. Based on the selection criteria specified in the WHERE clause, the query returns values into the host variables name_v and city_v. Queries can return multiple rows depending on the selection criteria.

NOTE: If a query can return more than one row, do not use the INTO clause; instead use a cursor operation, described in the "Introduction to Cursors" section.

7.1.2 INPUT Host Variables in Query Statements

Host variables that are used as input variables in a SQL statement are referred to as *input host variables*. In a SELECT statement, you can use INPUT host variables in the WHERE clause. The example in the "Elements of a Query" section uses the input host variable cust_no_v in the WHERE clause.

7.1.3 OUTPUT Host Variables in Query Statements

Host variables that are used for receiving query results are referred to as *output host variables*. In a query statement you use output host variables in the INTO clause. The example in the "Elements of a Query" section uses the output host variables name_v and city_v in the INTO clause. Whenever you use output host variables in a SELECT statement, the number of output host variables must equal the number of columns specified in the SELECT list.

7.2 Queries Returning a Single Row

The simplest queries are those that return a single row. When you know that a query is to return a single row, use the INTO clause to receive the result of the query.

EXAMPLE

The following example shows a query against the customer table to retrieve the columns last_name, city, and state:

```
/*
** Connect to the default database
*/
EXEC SQL CONNECT TO DEFAULT ;

EXEC SQL
     SELECT last_name, city, state
     INTO :name_v, :city_v, :state_v
     FROM customer
     WHERE cust_no = 1001 ;

if (sqlca.sqlcode < 0)
{
     printf ("Select statement failed (%ld : %s)\n",
             sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}
/*
** Successful select. Report results.
*/ printf ("last_name : %s, city : %s, state : %s\n",
     name_v, city_v, state_v);
EXEC SQL COMMIT WORK ;

/*
** Disconnect from the default database */
*/
EXEC SQL DISCONNECT DEFAULT ;
```

NOTE: If you use SELECT with an INTO clause in a query that returns multiple rows, the operation returns an error message.

7.3 Queries Returning Multiple Rows

When you design a query that can return more than one row, you must use a *cursor* in the SELECT statement. The following sections introduce cursors and explain how to use them.

7.3.1 Introduction to Cursors

A cursor is an SQL object that you associate with a specific SELECT statement. You associate a named cursor with a SELECT operation by declaring the cursor. To access the rows corresponding to the SELECT operation, follow these general steps:

1. Use a DECLARE statement to associate a cursor with the query
2. OPEN the cursor
3. Use FETCH repeatedly on the opened cursor to retrieve all the rows in the result set
4. CLOSE the cursor

You open a cursor with the OPEN statement. When you open a cursor, the query associated with the cursor executes, identifying the result set. This result set is referred to as the active set. An open cursor always points to a row in the active set; this row is referred to as the current row.

Use the FETCH statement to retrieve rows from an active set. Each time you execute a FETCH, the cursor moves by one row in the active set. A cursor can be in one of the two states: open or closed. When a cursor is in the open state, it is associated with an active set. It can point before the first row, at the current row, or after the last row. When the query operation is complete, you must close the cursor with a CLOSE statement. When a cursor is in the closed state, the cursor is no longer associated with the active set, although it remains associated with the SELECT statement.

The following sections explain the use of cursors for processing rows returned by a SELECT statement.

7.3.2 Associating a Cursor with a Query

Use the DECLARE CURSOR statement to associate a cursor with a SELECT statement. The DECLARE CURSOR statement declares a cursor by assigning it a name and associating it with a SELECT statement.

This is the syntax for a DECLARE CURSOR statement:

SYNTAX

```
EXEC SQL
      DECLARE cursor_name CURSOR FOR SELECT ... FROM ... ;
```

EXAMPLE

The following code fragment declares the cursor cust_cur for the associated SELECT statement:

```
EXEC SQL
   DECLARE cust_cur CURSOR FOR
      SELECT last_name, city, state
      FROM customer
      WHERE cust_no = :cust_no_v;
```

The DECLARE CURSOR statement is a declarative SQL statement. ESQL cannot interpret a reference to a cursor that has not been declared. You must declare the cursor before invoking any other SQL-92 statement that references the cursor. A cursor declared in one ESQLC source file cannot be referred to in another ESQLC source file. A cursor name used in a DECLARE CURSOR statement must be unique in a program source file.

7.3.3 OPEN a Cursor

You open a cursor with the OPEN statement. When you open a cursor, the database executes the SELECT statement associated with the cursor. The rows retrieved as a result of the execution form the active set.

This is the syntax for an OPEN statement:

SYNTAX

```
EXEC SQL OPEN cursor_name ;
```

EXAMPLE

The following code fragment illustrates how to use an OPEN statement:

```
EXEC SQL
     DECLARE cust_cur CURSOR FOR
          SELECT last_name, city, state
          FROM customer
          WHERE cust_no = :cust_no_v;
EXEC SQL
     OPEN cust_cur ;
```

The OPEN statement sets the cursor to the open state. When the database processes an OPEN statement, it executes the statement using the current values of program variables. The cursor now points just before the first row of the resulting active set. While the cursor is in the open state, subsequent changes to any program variables that appear in the SELECT statement associated with the cursor do not affect the active set.

OPEN *cursor_name* does not use the values in associated input host variables after the cursor is opened. This means the result of the query is not affected if the variable values change after the cursor is opened, while it remains open. To reflect changes to host variables in query results, you must close and reopen the cursor.

7.3.4 FETCH Rows Using a Cursor

Use the FETCH statement to read the rows of the active set and return the values into host variables. The SELECT statement associated with the cursor does not include the INTO clause. With cursor operations, the INTO clause and the list of output host variables are in the FETCH statement.

This is the syntax for a FETCH statement:

SYNTAX

```
EXEC SQL
 FETCH cursor_name INTO :hostvar1 [, :hostvar2] , . . . ;
```

EXAMPLE

The following code fragment shows how to fetch rows opened by the cust_cur cursor:

```
/* Connect to the default database */
EXEC SQL CONNECT TO DEFAULT ;

/* Declare a cursor for retrieving columns from the customer table */
EXEC SQL
     DECLARE cust_cur CURSOR FOR
          SELECT last_name, city, state
          FROM customer
          WHERE cust_no = :cust_no_v;
/* Open the cursor */
EXEC SQL OPEN cust_cur ;

/* Fetch the query results into host variables */
for (;;)
{
     EXEC SQL
          FETCH cust_cur
          INTO :cust_no_v, :name_v, :city_v ;
     if (sqlca.sqlcode != 0) break ;
/*
** Process the result of a successful fetch here; still in the for loop.
*/
}
/*
** Process SQL_NOT_FOUND and error conditions.
*/
EXEC SQL CLOSE cust_cur ;
```

You must declare and open a cursor before you can fetch into it. The first time you execute a fetch, the database positions the cursor on the first row of the active set. This row is now the current row. Each subsequent fetch operation advances the cursor to the next row in the active set. The only way to return to a row that you fetched earlier is to close and reopen the cursor. The behavior of cursor operations is also dependent on the isolation level for the current transaction.

When the cursor is positioned on the last row of the active set or if the active set does not contain any rows, then a fetch operation returns the status SQL_NOT_FOUND in the SQLCA. When this is the status, you must first close and reopen the cursor before you attempt any further operations with it.

7.3.5 CLOSE a Cursor

The CLOSE cursor statement puts the cursor in the closed state. Once the cursor is closed, no FETCH calls can be issued on the cursor until it is reopened.

This is the syntax for the CLOSE cursor statement:

SYNTAX

```
EXEC SQL CLOSE cursor_name ;
```

EXAMPLE

The following code fragment shows how to use the CLOSE cursor statement to close the cursor cust_cur:

```
EXEC SQL
     CLOSE cust_cur ;
```

The active set of a closed cursor is undefined, and you cannot execute fetches against that cursor. After you close a cursor, the only valid statement referring to that cursor is an OPEN *cursor_name*.

7.3.6 UPDATE or DELETE the Current Row

ESQL allows UPDATE and DELETE operations on the row that a cursor is pointing to in the active set. You can implement these operations by using the CURRENT OF cursor construct in the WHERE clause of either an UPDATE or a DELETE statement.

This is the format of a WHERE clause with a CURRENT OF construct:

SYNTAX

```
WHERE CURRENT OF cursor_name ... ;
```

EXAMPLE

This code fragment illustrates how to use the CURRENT OF cursor construct in an UPDATE statement. The example updates the qty column of the current row, which is where the cursor is positioned in the active set:

```
EXEC SQL
     DECLARE ord_cur CURSOR FOR
          SELECT product, qty
          FROM orders
          WHERE order_no = :order_no_v
          FOR UPDATE OF qty ;
EXEC SQL OPEN ord_cur ;

EXEC SQL FETCH ord_cur INTO :product_v, :qty_v ;

EXEC SQL
     UPDATE orders
          SET qty = :qty_v + 1000
          WHERE CURRENT OF ord_cur ;
```

You can execute a *positioned update* only on an open cursor, and the cursor must be declared for a SELECT statement with a FOR UPDATE clause.

A *positioned delete* operation deletes the row that the cursor is currently positioned to in the active set. After a positioned delete operation, the cursor is positioned before the row immediately following the deleted row, or after the last row if no following row exists.

EXAMPLE

The following code fragment shows how to use positioned delete to delete the current row from the orders table:

```
EXEC SQL
     DECLARE ord_cur CURSOR FOR
          SELECT product, qty
          FROM orders
          WHERE order_no = :order_no_v ;

EXEC SQL OPEN ord_cur ;

EXEC SQL FETCH ord_cur INTO :product_v, :qty_v ;

EXEC SQL
     DELETE FROM orders WHERE CURRENT OF ord_cur ;
```

7.4 Array Fetches

As described in previous sections, the FETCH statement returns one row at a time from the active set selected by the OPEN statement. ESQL provides the *Array Fetch* facility for fetching multiple rows at a time from the active set into an explicitly declared array. This is more efficient for fetching a large number of rows since it greatly reduces the number of calls made to the database. See the "Using an SQLDA for Array Fetches" section in Chapter 10, "Dynamic SQL Management in ESQL-92," for a description of array fetches using a dynamically declared array.

NOTES

- The Array Fetch facility provided by Progress ESQLC is an extension to the SQL-92 standard.
- The Array Fetch feature which employs a C Language array for fetching multiple rows is unrelated to the ARRAY data type.

All ESQLC arrays (excluding character arrays) are mapped into a host language structure consisting of the target array and the current size of the array. This is the C Language structure form:

SYNTAX

```
struct new_type_name {
   long array_size;
   element_type_name actual_array[ constant_id ]
} ;
```

Host language statements can manipulate the array assuming that it is a structure with the same name as the array name and having two components *actual_array* and *array_size*. The *actual_array* component is the array and the *array_size* component contains the current size of the array.

EXAMPLE

The following example shows how to use two arrays to select up to fifty rows in one fetch:

(1 of 2)

```
/*
** 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 last_name, cust_no FROM customer ;

if (sqlca.sqlcode != 0) goto err ;

EXEC SQL OPEN customer_cursor ;

if (sqlca.sqlcode != 0 ) goto err ;
```

(2 of 2)

```
for (;;)
{
     int  i ;

     EXEC SQL FETCH customer_cursor
     INTO :customer_name_array, :customer_id_array ;

     /*
     ** Note that with array fetches, one or more rows
     ** could have been returned by the current execution of the
     ** FETCH statement even if the status code returned is
     ** SQL_NOT_FOUND.
     */

     if ((sqlca.sqlcode != 0)  && (sqlca.sqlcode != SQL_NOT_FOUND))
          break ;
     for (i = 0 ; i < customer_name_array.tpe_size ; i++ )
          {
               printf ("Customer id = %ld Customer Name : %s\n",
               customer_id_array.tpe_array[i],
               customer_name_array.tpe_array[i]) ;
          }

     if (sqlca.sqlcode != 0) break ;
} /* end for (;;) */
if (sqlca.sqlcode != SQL_NOT_FOUND) goto err ;

EXEC SQL CLOSE customer_cursor ;

if (sqlca.sqlcode != 0) goto err ;

EXEC SQL COMMIT WORK ;
if (sqlca.sqlcode < 0)
     printf ( "COMMIT WORK returned error %ld\n", sqlca.sqlcode) ;
return ;

err:
     EXEC SQL ROLLBACK WORK ;
     return ;
```

Executing the FETCH statement sets `tpe_size` to the actual number of rows returned.

NOTE: The array fetch facility provided by ESQL is an extension to the SQL-92 standard.

8

Handling NULL Values in ESQL-92

This chapter describes how to handle NULL values in ESQL. It discusses inserting and retrieving NULL values and how to manage NULL values in clauses of a SELECT statement.

8.1 Introduction

Use NULL values in a database when the value in a column is unknown or when the value is not applicable. For example, if an employee has not been assigned to a department, you might set the department number in the corresponding row to a NULL value.

To illustrate further, suppose the commission column of the employee table is applicable only to sales persons. This column could contain a NULL value for employees from all other departments. Note that a numeric zero is not the same as a NULL value. Similarly, a string of blanks is not the same as a NULL value.

In Progress SQL-92, the default value for a column is NULL unless the column definition contains the DEFAULT clause. A column of any data type can have a NULL value. You can specify that a column of any type not have NULL values by using the NOT NULL clause in the CREATE TABLE statement. Using NULL values is discussed in the following sections.

8.2 INSERT NULL Values

Insert a NULL value into a column when a column value is not known or not applicable. You can insert a NULL into a column:

- By default
- By using the NULL keyword
- By using indicator variables

The following sections discuss each of these three ways of inserting a NULL value.

8.2.1 INSERT Null Values by DEFAULT

In Progress SQL-92, the default value for a column is NULL if the column definition does not contain the DEFAULT clause.

EXAMPLE

The following CREATE TABLE example defines two columns with the NOT NULL specification:

```
CREATE TABLE employee (
     empno     INTEGER NOT NULL,
     ename     CHAR(10),
     job       CHAR(10),
     hiredate  DATE,
     sal       NUMERIC(10,2),
     commission  NUMERIC(10,2),
     deptno    INTEGER NOT NULL,
     projno    INTEGER
     ) ;
```

For example, to insert a row for a new employee the INSERT statement could be:

```
EXEC SQL
     INSERT INTO employee (empno, ename, job, hiredate, sal, deptno)
     VALUES (8585, 'RALPH', 'CLERK', '02/02/1999', 12000, 10) ;
```

When you execute this INSERT statement, the database inserts a NULL VALUE into the commission and projno columns.

You can restrict any type of column not to accept a NULL value by specifying NOT NULL in the CREATE TABLE statement for that column. If you create a unique index for a column, it can have at most one row with a NULL value in that column.

8.2.2 Use the NULL Keyword

You can use the NULL keyword in the value list to indicate that a column in the column list should be assigned a NULL value.

EXAMPLE

The following example shows how to insert NULL values for the commission and projno columns using the NULL keyword:

```
EXEC SQL
     INSERT INTO employee
     VALUES (8585, 'RALPH', 'CLERK', '02/02/1999',
     2000, NULL, 10, NULL) ;
```

8.2.3 Use Indicator Variables

You can use indicator variables to insert NULL values into a column. Set the indicator variable for a particular column to negative one (-1) and use the indicator variable along with the associated host variable in the value list of the INSERT statement.

EXAMPLE

The following example shows how to insert NULL values for the commission and projno columns in the employee table using indicator variables:

```
/*
** Declare section for host variables and indicator variables>
*/
EXEC SQL BEGIN DECLARE SECTION ;
   .
   .
   .
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
empno_v = 2402 ;
strcpy (ename_v, 'RALPH') ;
strcpy (job_v, 'CLERK') ;
strcpy (hiredate_v, '02/02/1999') ;
sal_v = 2000 ;
commission_i = -1 ;
deptno_v = 10 ;
projno_i = 1 ;   .

EXEC SQL
     INSERT INTO employee
     VALUES (:empno_v, :ename_v, :job_v, :hiredate_v,
             :sal_v, :commission_v INDICATOR :commission_i,
             :deptno_v, :projno_v:projno_i) ;
```

This example illustrates setting the indicator variable to negative one (-1), rather than placing the NULL keyword in the INSERT statement. The program then inserts a NULL value, not the value in the host variable.

8.3 UPDATE with NULL Values

You can modify a column value to contain a NULL value by using the UPDATE statement. Assign the desired column the keyword value NULL in an UPDATE statement.

EXAMPLE

The following example shows how to update the projno column to NULL, indicating there is no project assigned to employee 2040:

```
EXEC SQL
     UPDATE employee
     SET projno = NULL
     WHERE empno = 2040 ;
```

You can also modify a column value to contain a NULL by using indicator variables. Set the indicator variable to negative one (-1) and use it with its associated host variable.

EXAMPLE

The following code fragment shows how to update the employee table using an indicator variable:

```
projno_i = -1 ;
EXEC SQL
     UPDATE employee
     SET projno = projno_v:projno_i
     WHERE empno = 2040 ;
```

8.4 RETRIEVE NULL Values

You can manage the retrieval of NULL values in the following ways with a SELECT statement:

- Use an indicator variable
- Use the scalar function NVL

Following sections describe how to recognize NULL values when you retrieve them.

8.4.1 Using Indicator Variables

After a query, you can identify NULL values by examining the indicator variables associated with the target output host variables.

EXAMPLE

The following example shows how to identify NULL values after retrieving a row:

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

8.4.2 Using the NVL Scalar Function

You can use the scalar function NVL to identify NULL values in a column. The syntax of the NVL function is:

SYNTAX

```
EXEC SQL
  NVL (expression1, expression2 ) ;
```

If the first expression value **is not NULL**, the NVL function returns its value. If the first expression value **is NULL**, NVL returns the value of the second expression. The following example shows how to use NVL to identify NULL values in the commission column of the employee table:

```
EXEC SQL
     SELECT sal + NVL (commission, 0) 'Total Salary'
     FROM employee ;
```

If the value of the commission column is NULL, the function adds zero to the base salary of the employee and returns the total in a temporary column named Total Salary.

8.5 NULL Values in Expressions

You can use expressions in the SELECT list for retrieving NULL values. If any value that participates in an arithmetic expression is NULL, the value of the entire expression is NULL, as this example illustrates:

```
EXEC SQL
     SELECT ename, sal, sal + commission
     FROM employee
     WHERE empno = 2452 ;
```

In this example, if the commission for empno 2452 is NULL, then the value of the expression sal + commission in the SELECT list evaluates to NULL.

8.6 NULL Values in the WHERE Clause

A select operation succeeds if the search condition in the WHERE clause evaluates to TRUE. A row is **not** selected if the search condition evaluates to FALSE or is unknown.

EXAMPLE

In the following example, if commission is NULL for empno 2004, the search condition is **not** satisfied:

```
WHERE ((sal + commission) < 5000)
     AND empno = 2004 ;
```

The search condition with an OR evaluates to TRUE:

```
WHERE ((sal + commission) < 5000)
     OR empno = 2004 ;
```

You can select or reject a row containing NULL values in a column by using the [NOT] NULL search condition:

```
... WHERE commission IS NOT NULL ;
... WHERE commission IS NULL ;
```

EXAMPLE

The following example shows how to select employee names and department numbers for employees who have not been assigned to any project:

```
EXEC SQL
     SELECT ename, deptno
     FROM employee
     WHERE projno IS NULL ;
```

You perform a join between two tables using a WHERE clause. Rows do not meet the search criteria if either the salary column in either table is NULL or if the salary columns in both tables are NULL:

```
WHERE employee.salary = title.salary ;
```

8.7 NULL Values in a GROUP BY Clause

When you apply a GROUP BY clause to a column containing NULL values, all NULL values are placed into the same group.

EXAMPLE

In the following example the SELECT statement returns a count and a list of employee names (ename). All the employees who are not assigned to a project (projno) are grouped together in the result set:

```
EXEC SQL
     SELECT ename, COUNT (*) FROM employee
     GROUP BY projno ;
```

8.8 NULL Values in an ORDER BY Clause

When you apply an ORDER BY clause to a column containing NULL values, the NULL value is treated as being greater than any non-NULL value. When the ordering is ascending, the NULL values sort at the end. When the ordering is descending, the NULL values sort first.

EXAMPLE

The following example shows the selection of employees (ename) ordered by the project numbers (projno). The ORDER BY clause specifies descending (DESC). The employees not assigned to a project are reported first, and those with the highest project numbers are reported last:

```
EXEC SQL
     SELECT ename, deptno FROM employee
     ORDER BY projno DESC ;
```

8.9 NULL Values in Scalar Functions

Most scalar functions return a NULL value if any of the arguments evaluate to NULL. Scalar functions like NVL that specifically check for a NULL value are exceptions.

8.10 NULL Values in Aggregate Functions

Aggregate functions ignore rows with NULL values in an argument and return the value based on the rest of the rows. You can specify the COUNT function with an expression as the argument.

EXAMPLE

For example, to count all the rows in the employee table, including those with NULL values in the commission column, the SELECT statement is:

```
EXEC SQL
     SELECT COUNT ( NVL (commission, 0))
     FROM employee ;
```

If the column contains only NULL values, the COUNT DISTINCT *column_name* returns zero and the rest of the aggregate functions return NULL for that column. An exception is when you specify COUNT(*). This returns the number of rows regardless of the column values.

9

Handling Errors in ESQL-92

This chapter describes handling errors in ESQL. Specifically, it discusses:

- Evaluating the SQL Communication Area (SQLCA)
- Using the WHENEVER statement
- Evaluating indicator variables

9.1 Using an SQLCA for Error Handling

The *SQL Communication Area* (SQLCA) is a mechanism for returning the status of an SQL statement execution in an ESQL program. For example, the SQLCA reports an error if a SELECT statement has a reference to a table that does not exist in the database.

In ESQLC the SQLCA is a C Language structure. The ESQL precompiler automatically declares and defines a global SQLCA. The components of the SQLCA structure report information on the status of the execution of an SQL statement.

9.2 The SQLCA Structure

The complete SQLCA structure is in the `sql_lib.h` file, in the include directory of your installation. This is an excerpt from the SQLCA structure:

```
struct sqlca {
    char   sqlcaid[8] ;
    long   sqlcabc ;
    long   sqlcode ;
    unsigned long sqlerrml ;
    char   sqlerrm[512] ;
    char   sqlerrp[8] ;
    unsigned long sqlerrd[8] ;
    char   sqlwarn[8] ;
    char   sqlext[8] ;
        .
        .
        .
    long   number ;
    char   more[1] ;
    char   command_function[128] ;
    char   dynamic_function[128] ;
    long   condition_number ;
    char   returned_sqlstate[6] ;
    char   class_origin[128] ;
    char   subclass_origin[250] ;
    char   table_name[128] ;
    char   column_name[128] ;
} ;
```

Descriptions of the fields in the SQLCA structure follow:

SQLCAID

Contains the string 'SQLCA'.

SQLCABC

Contains the size of the SQLCA structure.

SQLCODE

Gives the result of an SQL statement execution. The SQLCODE can have the following values:

- Zero: Indicates successful execution.
- Positive: Indicates successful execution with a status code. The database returns the positive status code SQL_NOT_FOUND when there are no more rows to be fetched. The SQL_NOT_FOUND status is defined in sql_lib.h.
- Negative: Indicates an error in execution of the SQL statement or a system failure.

SQLERRML

Contains the length of the error message in SQLERRM.

SQLERRM

Is a NULL terminated character string that is the error text corresponding to the code returned in the component SQLCODE.

SQLERRP

Currently not used.

SQLERRD

Is an array of six elements where each element is a long integer.

SQLERRD[2]

Indicates the number of rows processed after a successful execution of INSERT UPDATE and DELETE statements. SQLERRD[2] is set to the cumulative number of rows for a FETCH call associated with a cursor.

SQLERRD[3]

Is set to the number of rows fetched by the last FETCH call.

SQLWARN

A character array of size eight where each character value can either be a blank or a 'W'. A character 'W' in any of the character positions of the array indicates a warning during the execution of an SQL statement. Currently, only elements 0, 1, 2, 3, 4, and 6 of the array are used.

SQLWARN[0]

Set to 'W', then one or more of the other SQLWARN components have been set. If this component is blank no warning has been set.

SQLWARN[1]

Set to 'W' to indicate that one or more of the strings returned by the previous FETCH call have been truncated. Use the indicator variables to determine which of the strings were truncated.

SQLWARN[2]

Set to 'W' when one or more NULL values were ignored in the computation of an aggregate function.

SQLWARN[3]

Set to 'W' to indicate that the number of items in the SELECT list does not equal the number of host variables in the INTO clause; the query returns all the elements it can, which is the lesser of the two counts.

SQLWARN[4]

Set to 'W' after a successful execution of an UPDATE or a DELETE statement without a WHERE clause; it is more useful to check for this component while working with dynamic statements.

SQLWARN[6]

Set to 'W' if the transaction is implicitly marked for ROLLBACK. The application must roll back the current transaction before executing the next SQL statement.

SQLEXT

Currently not used.

9.2.1 Using an SQLCA to Check for Errors

Sometimes an attempted SQL operation does not succeed. You can check the SQLCA for the error code and the corresponding error message. Use the following components of the SQLCA structure to obtain the error codes and error messages:

- SQLCODE
- SQLERRM
- SQLERRML
- SQLERRD

The SQLCODE component indicates the return status after the execution of an SQL statement. SQLCODE is set to zero for a successful execution and is negative for a failure. Also, SQLCODE can report SQL_NOT_FOUND, which is set during a FETCH operation when there are no more rows to be fetched.

The component SQLERRM is a null-terminated character string, which is the diagnostic text corresponding to the SQLCODE.

The component SQLERRML contains the length of the error message in SQLERRM.

The component SQLERRD is an array of six INTEGER variables.

EXAMPLE

The following code fragment shows how to use the SQLCA components SQLCODE and SQLERRM:

```
/*
** CONNECT TO database ESQL statements go here
*/

EXEC SQL BEGIN DECLARE SECTION ;
     long deptno_v ;
     char dname_v [20] ;
     char loc_v [10] ;
EXEC SQL END DECLARE SECTION ;

/*
** Determine values for input host variables
*/
deptno_v = 30 ;
strcpy (dname_v, "ACCOUNTS") ;
strcpy (loc_v, "BOSTON") ;

EXEC SQL
     INSERT INTO department (deptno, dname, loc)
     VALUES (:deptno_v, :dname_v, :loc_v) ;

if (sqlca.sqlcode < 0)
{
     printf("Insert statement failed (%ld : %s)\n",
             sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;

     /*
     ** DISCONNECT here
     */
     exit (1);
}

/*
** Successful INSERT; commit changes
*/
EXEC SQL COMMIT WORK ;

printf ("Inserted one row \n\n");

/*
** DISCONNECT here
*/
```

EXAMPLE

The following example shows how to check for the SQL_NOT_FOUND status code after a FETCH operation:

(1 of 2)

```
EXEC SQL CONNECT TO DEFAULT ;

/*
** Declare a cursor for retrieving customer information
*/
EXEC SQL
     DECLARE cust_cur CURSOR FOR
     SELECT last_name, city, state
     FROM customer
     WHERE cust_no = :cust_no_v;
if (sqlca.sqlcode < 0)
{
     print ("Declare cursor statement failed (%ld : %s)\n",
     sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}
/*
** Open the cursor
*/
EXEC SQL OPEN cust_cur ;

if (sqlca.sqlcode < 0)
{
     printf ("Open cursor statement failed (%ld : %s)\n",
          sqlca.sqlcode, sqlca.sqlerrm);
     EXEC SQL ROLLBACK WORK ;
     EXEC SQL DISCONNECT DEFAULT ;
     exit (1);
}

/*
** Fetch rows and return result values into host variables
** Break out of for loop when no more rows
*/
for (;;)
{
     EXEC SQL FETCH cust_cur INTO :cust_no_v, :name_v, :city_v ;

     if (sqlca.sqlcode == SQL_NOT_FOUND) break ;
```

(2 of 2)

```
 if (sqlca.sqlcode < 0)
     {
          printf ("FETCH cursor statement failed (%ld : %s)\n",
               sqlca.sqlcode, sqlca.sqlerrm);
          EXEC SQL CLOSE cust_cur ;
          EXEC SQL ROLLBACK WORK ;
          EXEC SQL DISCONNECT DEFAULT ;
          exit (1);
     }

/*
** Successful fetch. Print results, continue the for loop.
*/
printf ("cust_no : %d, last_name : %s, city : %s\n",
     cust_no_v, name_v, city_v) ;
}

EXEC SQL CLOSE cust_cur ;
EXEC SQL COMMIT WORK :
EXEC SQL DISCONNECT DEFAULT ;
```

9.2.2 Using an SQLCA to Check for Warnings

You can use an SQLCA to check for warnings that occur during the execution of an SQL statement. The component SQLWARN of SQLCA is an array of eight characters. Each element of the array can be either a blank or a `W'.

Currently Progress uses the elements 0, 1, 2, 3, 4, and 6 of the SQLWARN array. These are the conditions under which warning flags are set:

- One or more string values returned by a query are truncated
- One or more NULL values are ignored in the computation of an aggregate function
- The number of items in the SELECT list does not equal the number of host variables in the INTO clause
- The database implicitly marks the transaction for ROLLBACK

EXAMPLE

The following example computes the average commission for employees in the sales department, and shows how to evaluate the SQLWARN array to check for null values in the data:

```
EXEC SQL BEGIN DECLARE SECTION ;
     FLOAT  comm_v ;
EXEC SQL END DECLARE SECTION ;
   .
   .
    .
EXEC SQL CONNECT TO DEFAULT ;

EXEC SQL
     SELECT AVG (commission)
     INTO :comm_v
     FROM employee
     WHERE deptno = 20 ;

if (sqlca.sqlcode == 0)
{
     printf ("commission : %d", comm_v) ;
     /*
     ** Select statement completed; also check SQLCA for warnings
     */
     if (sqlca.sqlwarn[2] == 'W')
     {
          printf ("One or more NULL values were ignored\n") ;
          printf ("in the computation of average commission.\n") ;
     }
}
```

This example evaluates the SQLWARN[2] component of the SQLCA to determine if any NULL values were ignored in the computation of the average commission.

9.3 Using the WHENEVER Statement for Error Handling

The WHENEVER statement is a declarative statement that defines the actions to take when there is an error. The WHENEVER statement is easy to use, flexible, and reduces the code size of the application program.

This is the syntax for a WHENEVER statement:

SYNTAX

```
WHENEVER specific_exception specific_action ;
```

The following sections contain more information about the WHENEVER statement.

9.3.1 Branching Under Exceptions

The main advantage of the WHENEVER statement is its capability to branch under exceptions. The syntax for the WHENEVER statement shows how you specify branching in response to an exception condition:

SYNTAX

```
EXEC SQL
   WHENEVER { NOT FOUND | SQLERROR | SQLWARNING }
   { { GOTO host_language_label | GO TO host_language_label
        | STOP | CONTINUE } } ;
```

These are the exception conditions for which you can specify branching:

`NOT FOUND`

The exception `NOT FOUND` is set when `SQLCODE` is set to `SQL_NOT_FOUND`.

`SQLERROR`

The exception `SQLERROR` is set when `SQLCODE` is set to negative.

SQLWARNING

The exception SQLWARNING is set when SQLWARN[0] is set to 'W'.

GOTO | GO TO *host_language_label*

Specifying the GOTO *host_language_label* or GO TO *host_language_label* causes control to pass to the statement at that label.

STOP

STOP terminates the program.

CONTINUE

CONTINUE ignores the SQLCA status and control passes to the next statement in the program.

EXAMPLE

The following code fragment uses the WHENEVER statement for exception handling in an ESQL program that deletes a row:

```
EXEC SQL CONNECT TO DEFAULT ;

/*
** At a SQLERROR or NOT FOUND exception, branch to label do_rollback
*/

EXEC SQL WHENEVER SQLERROR GOTO do_rollback ;

EXEC SQL WHENEVER NOT FOUND GOTO do_rollback ;

/*
** Assign a value to cust_no_v
*/
cust_no_v = 1002 ;
EXEC SQL
     DELETE
     FROM customer
     WHERE cust_no = :cust_no_v ;

/*
** Successful delete; Commit work, disconnect, go home
*/
EXEC SQL COMMIT WORK ;
EXEC SQL DISCONNECT DEFAULT ;
exit (0) ;

do_rollback:
if (sqlca.sqlcode == SQL_NOT_FOUND)
     printf ("Customer number not found in table\n");
else
     if (sqlca.sqlcode < 0)
     {
          strncpy (errmesg, sqlca.sqlerrm, sqlca.sqlerrml);
          errmesg [sqlca.sqlerrml] = '\0' ;
          printf ("Error : %s\n", errmesg);
     }

EXEC SQL WHENEVER SQLERROR CONTINUE ;

EXEC SQL ROLLBACK WORK ;

EXEC SQL DISCONNECT DEFAULT ;
exit (1) ;
```

You should specify CONTINUE in the WHENEVER statement in the exception handling code. This prevents the WHENEVER statement from passing control to the same label, which results in a program loop if any of the SQL statements within the label fail.

Use either CONTINUE or GOTO with the WHENEVER statement, rather than STOP. Although STOP is valid, using it terminates the program without any final reporting.

Handle the SQL_NOT_FOUND Condition

When you issue a FETCH operation and there are no more rows to be fetched, the database sets SQL_NOT_FOUND in the SQLCA. When your application evaluates the SQLCA and finds SQL_NOT_FOUND, your application should close the cursor.

An UPDATE or DELETE operation can return the SQL_NOT_FOUND when no rows are updated or deleted because there are no rows to satisfy the conditions in the WHERE clause.

Using WHENEVER with Explicit Error Checking

In combination with the WHENEVER statement, your application can explicitly evaluate the SQLCA.

EXAMPLE

The code fragment in the next example illustrates explicitly evaluating the SQLCA:

```
EXEC SQL
     WHENEVER SQLERROR GOTO do_rollback ;
/*
** Fetch rows and return result values into host variables
*/
for (;;)
{
     EXEC SQL FETCH cust_cur INTO :cust_no_v, :name_v, :city_v ;
     if (sqlca.sqlcode == SQL_NOT_FOUND)
          break ;
}
```

9.3.2 Scope of the WHENEVER Statement

The scope of a WHENEVER statement begins when control passes to the statement. The WHENEVER test for a condition remains in effect until there is another WHENEVER statement for the same exception, or until the end of the program.

9.4 Using Indicator Variables for Error Handling

You can use indicator variables in an ESQL application program to detect whether the retrieved value is NULL or has been truncated. The value is truncated when the length of the returned value is greater than the length of the target host variable.

See Chapter 8, "Handling NULL Values in ESQL-92," for more information on evaluating indicator variables.

These are the values for indicator variables and their meanings:

- 0

  The returned value is **NOT NULL** and has not been truncated. The corresponding host variable contains the retrieved value.

- -1

  The returned value **IS NULL** and the value of the corresponding host variable is not defined.

- > 0

  The returned value was truncated because the length of the host variable was too small. The > 0 value in the indicator variable is the actual length of the returned value before it was truncated.

EXAMPLE

The following code fragment shows how to evaluate indicator variables to detect null:

```
EXEC SQL
     SELECT ename, deptno, commission
     INTO :ename_v, :deptno_v, :commission_v:commission_i
     FROM employee
     WHERE empno = 2002 ;
if (commission_i == -1)
     printf ("No commission recorded for employee %s ", ename_v) ;
```

This example evaluates the indicator variable commission_i to determine if the retrieved value for the commission column is NULL.

10

Dynamic SQL Management in ESQL-92

This chapter describes dynamically defined statements that you use to write flexible ESQL applications. Specifically, it describes:

- Dynamic non-SELECT statements
- Dynamic SELECT statements
- Using an SQLDA
- DESCRIBE statements

10.1 Introduction

When you write an ESQL application and you place specific SQL statements in your program, these statements are called *static SQL statements*. For example, an application might produce a daily inventory report, executing the same query every morning. However, in many applications the exact SQL statements are not known at compile time. Part or all of the SQL statements are known only at run time, often from user input provided to the application. Such SQL statements are called *dynamic SQL statements*. For example, the SQL statements you address to the SQL Explorer tool are ad hoc statements. They might be, for example, SELECT or INSERT. These statements are processed by SQL Explorer as dynamic SQL statements. Dynamic SQL statements provide more flexibility in what they can accomplish, but the run-time cost is usually higher.

These are the general steps to follow when you execute a dynamic statement that is a **SELECT** statement:

1. PREPARE the SELECT statement.
2. DECLARE a cursor.
3. OPEN the cursor.
4. FETCH rows using the cursor.
5. CLOSE the cursor.

These are the general steps to follow when you execute a dynamic statement that is a **non-SELECT** statement:

1. PREPARE the statement.
2. EXECUTE the statement.

10.2 PREPARE Statement

Application programs use a *PREPARE* statement to prepare another SQL statement for execution. When using PREPARE, the SQL engine parses the SQL statement to check for syntax errors. Then the engine assigns a statement identifier to the SQL statement.

An SQL statement that you PREPARE can be specified either as a character string or as a reference to a host variable. If the SQL statement is specified using a host variable reference, the host variable must be a character array.

PREPARE statements can be used for both SELECT statements and other statements. The SQL statement is prepared once, but is executed as often as necessary within the same transaction. If you COMMIT or ROLLBACK the current transaction and you want to re-execute the SQL statement, you must PREPARE it again.

EXAMPLE

The following code fragment shows how to use a PREPARE statement:

```
EXEC SQL BEGIN DECLARE SECTION ;
     char sql_str [256] ;
EXEC SQL END DECLARE SECTION ;

strcpy (sql_str, "delete from customer where cust_no = :p1") ;

EXEC SQL PREPARE delstmt FROM :sql_str ;
```

The statement identifier used in the PREPARE statement is delstmt. This identifier is used for references to the statement in other ESQL statements. The host variable, sql_str, is declared in the DECLARE SECTION.

The host variable reference made in the SQL statement, such as p1, is a dynamic parameter. The name of the host variable used in the prepared SQL statement is not related to the name of the host variable that would contain the corresponding value. For example, in the prepared SQL statement string, you might use the name p1, and the name of the host variable used to supply the value might be cust_no_v.

EXAMPLE

The following example shows how to use a character string in the PREPARE statement:

```
EXEC SQL
   PREPARE selstmt FROM "select * from customer" ;
```

Note that the dynamic SQL statements must not contain the terminating semicolon. The SQL statements that you **cannot** process with a PREPARE statement are:

- CLOSE
- DECLARE
- EXECUTE
- FETCH

- OPEN
- PREPARE
- SELECT with an INTO clause

You can execute other dynamic SQL statements using a two-step process. The first step is to prepare the statement with the PREPARE statement. The second step is to execute the prepared statement by supplying values, if any, to the input host variables specified in the PREPARE statement. This second step uses the EXECUTE statement.

10.3 EXECUTE Statement

The EXECUTE statement takes the statement identifier of a prepared statement and executes it.

EXAMPLE

The following example shows how to use the EXECUTE statement in an ESQL program:

```
EXEC SQL BEGIN DECLARE SECTION ;
     char sql_str [256] ;
     long cust_no_v ;
EXEC SQL END DECLARE SECTION ;
/*
** Connect to the default database
*/
EXEC SQL CONNECT TO DEFAULT ;

strcpy (sql_str, "delete from customer where cust_no = :p1") ;

EXEC SQL PREPARE stmt FROM :sql_str ;

cust_no_v = 1001 ;

EXEC SQL EXECUTE stmt USING :cust_no_v ;
/*
** COMMIT the DELETE operation
*/
EXEC SQL COMMIT WORK ;
/*
** Disconnect from the default database.
*/
EXEC SQL DISCONNECT DEFAULT ;
```

The EXECUTE statement executes the prepared DELETE statement using the values supplied for each host variable. In this example, the host variable is cust_no_v. You PREPARE the SQL statement once, and execute the statement as often as necessary in the same transaction. If you COMMIT or ROLLBACK the current transaction and you want to execute an SQL statement again, you must PREPARE it again before you can execute it.

10.4 EXECUTE IMMEDIATE Statement

The *EXECUTE IMMEDIATE* statement is similar to using PREPARE and then EXECUTE. The EXECUTE IMMEDIATE statement accepts as input a character string or a host variable of character array type. A dynamic statement that will be executed with EXECUTE IMMEDIATE cannot contain any dynamic parameters.

EXAMPLES

The following examples show how to use the EXECUTE IMMEDIATE statement:

```
EXEC SQL BEGIN DECLARE SECTION ;
     char sql_str [256] ;
EXEC SQL END DECLARE SECTION ;

gets (sql_str) ;

EXEC SQL EXECUTE IMMEDIATE :sql_str ;
```

A character string can be used in place of a host variable reference, as follows:

```
EXEC SQL EXECUTE IMMEDIATE
     "delete from customer where cust_no = 1001" ;
```

The SQL string given to an EXECUTE IMMEDIATE statement should not have host variable references.

10.5 SELECT Statement

Dynamic SELECT statements are executed differently since they return data and information beyond the scope of the SQLCA. The statements used for dynamic SELECT include OPEN, FETCH, and CLOSE.

A simple form of a dynamic SELECT statement allows queries whose SELECT list is known but whose search criteria or ordering might vary. Follow these general steps to process a dynamic SELECT statement:

1. PREPARE the statement.
2. DECLARE a cursor.
3. OPEN the cursor.
4. FETCH rows using the cursor.
5. CLOSE the cursor.

10.5.1 OPEN a Cursor

The OPEN statement sets a cursor to the open state. When a cursor is opened, the SELECT statement associated with the cursor is executed producing a result set. This result set is referred to as the active set. Executing the SELECT statement uses the current program variables and leaves the cursor positioned just before the first row in the active set.

When the cursor is in the open state, any changes to the program variables of the SELECT statement do not change the rows retrieved in the active set.

EXAMPLE

The following example shows an OPEN operation on the cursor custcur:

```
/*
** Open the cursor
*/
EXEC SQL
     OPEN custcur ;
```

10.5.2 FETCH Rows Using a Cursor

The FETCH statement advances the cursor to the next row in the active set and retrieves the values from that row. When the FETCH statement is executed for the first time, the cursor is positioned on the first row, and all the column values of the first row are retrieved. Subsequent

calls to FETCH retrieve rows one by one from the active set. When all the rows in the active set are retrieved, SQL_NOT_FOUND is set in the SQLCA to indicate that all rows have been retrieved.

EXAMPLE

The following example shows how to use the FETCH statement:

```
/*
** CONNECT to the DEFAULT database
*/
EXEC SQL CONNECT TO DEFAULT ;

EXEC SQL
     PREPARE cust_sel_stmt
          FROM "SELECT cust_no, last_name, city FROM customer" ;

EXEC SQL
     DECLARE cust_cursor CURSOR FOR cust_sel_stmt ;

/*
** Open the DECLARED cursor cust_cursor
*/
EXEC SQL OPEN custcur ;
/*
** FETCH rows and print if successful fetch
*/
for (;;)
{
     EXEC SQL
         FETCH cust_cursor
          INTO :cust_no_v, :name_v, :city_v ;

     if (sqlca.sqlcode != 0)
          break;

     printf ("%d, %s, %s\n", cust_no_v, name_v, city_v) ;
}

/*
** CLOSE the cursor, issue COMMIT to free resources
** and DISCONNECT from the database.
*/
EXEC SQL CLOSE cust_cursor ;
EXEC SQL COMMIT WORK ;
EXEC SQL DISCONNECT DEFAULT ;
```

10.5.3 CLOSE a Cursor

Use the CLOSE statement to close a cursor. The cursor then becomes invalid until it is opened again using the OPEN statement. The following statement closes the cursor, cust_cursor:

```
EXEC SQL CLOSE cust_cursor ;
```

10.6 SQL Descriptor Area (SQLDA)

The SQL Descriptor Area (SQLDA) is a storage area for descriptive information about a dynamic SQL statement. An SQLDA can be used while passing parameter values for executing a dynamic SQL statement and while retrieving results of a dynamic SELECT statement.

Dynamic SELECT statements must be executed differently from non-SELECT statements, since they return information and data beyond the scope of the SQLCA. A program executing a dynamic SELECT statement must determine the lengths and types of values being returned.

Similarly, a program executing a dynamic statement that is a non-SELECT statement must know the input values supplied to the parameters referenced in the statement.

For both categories of statements, determine the required value with a DESCRIBE statement. The DESCRIBE statement obtains information about a prepared statement. For more information on the DESCRIBE statement, see the "DESCRIBE Statement" section.

10.6.1 When to Use an SQLDA

The SQLDA can be used to:

- Hold information about the input host variables that contain input values for a dynamic SELECT or non-SELECT statement. The SQLDA used for this purpose is an input SQLDA. You use an input SQLDA in an EXECUTE statement or in an OPEN cursor statement corresponding to a dynamic SELECT statement.
- Hold information about the output host variables where the values are to be returned by a dynamic SELECT statement. The SQLDA used for this purpose is an output SQLDA.

Use an OUTPUT SQLDA for the following types of statements:

- DESCRIBE statement associated with a dynamic SELECT statement.
- FETCH statement associated with a dynamic SELECT statement. In ESQLC, an SQLDA is implemented as a structure. You must explicitly declare an SQLDA before using it in an ESQL program.

10.6.2 SQLDA Structure

The complete SQLDA structure is in the `sql_lib.h` file in the `include` directory of your installation. Here is the SQLDA structure:

(1 of 2)

```
struct sqlda {
  dh_i32_t      sqld_size;              /* da, DEPRECATED, max # cols in
                                        da */
  dh_i32_t      sqld_nvars;             /* da, DEPRECATED, actual/desired
                                        # cols */
  void          **sqld_varptrs;         /* col, ptr to data buffer */
  dh_u32_t      *sqld_lengths;          /* col, size (bytes) of data in
                                        col as bin (+1 char if char data)
                                        */
  dh_i32_t      *sqld_types;            /* col, data type */
  dh_u32_t      *sqld_precision;        /* col, size (chars) of data in
                                        col as char */
  dh_i16_t      *sqld_scale;            /* col, decimal digits to right of
                                        DP */
  dh_u32_t      *sqld_flags;            /* col, allows nulls */
  dh_i32_t      **sqld_ivarptrs;        /* col, ptr to indicator data (sz,
                                        null) */
  dh_i16_t      sqld_varnmsize;         /* da, max size (bytes) data col
                                        name */
  dh_i16_t      sqld_indvarnmsz;        /* da, DEPRECATED, max size
                                        (bytes) data indicator name */
  dh_char_t     **sqld_varnames;        /* col, ptr to user's data col
                                        name */
  dh_char_t     **sqld_indvarnames;     /* col, ptr to user's indicator
                                        name */
  dh_u32_t      sqld_arraysz;           /* da, # rows in multi-row
                                        operation */
  dh_u32_t      sqld_clengths;          /* col, actual data size in chars
                                        */
  dh_u32_t      *sqld_returned_lengths; /* col, Dictionary size of col */
  dh_u32_t      *sqld_odbc_flags;       /* col, ODBC flags: AUTO, sIGN,
                                        UNNAMED, etc */
  dh_i16_t      *sqld_date_int_code;    /* col, DEPRECATED, internal date
                                        code */
  dh_i16_t      *sqld_date_int_prec;    /* col, DEPRECATED, # seconds
                                        digits to right of DP*/
  dh_char_t     **sqld_collat_name;     /* col, collation sequence name */
  dh_char_t     **sqld_charset_name;    /* col, charset name */
```

(2 of 2)

```
dh_i32_t      sqld_max_size;            /* da, DEPRECATED, max # cols in
                                        da */
dh_char_t     **sqld_base_column_name;  /* col, base column name */
dh_char_t     **sqld_base_table_name;   /* col, base table name */
dh_i16_t      *sqld_searchable;         /* col, col is searchable (WHERE
                                        clause) */
dh_i16_t      *sqld_updatable;          /* col, col is updatable (INSERT,
                                        UPDATE, etc.) */
dh_i16_t      *sqld_param_type;         /* col, param is IN, OUT, or
                                        INTOUT */
dh_char_t     **sqld_table_name;        /* col, table name */
dh_i16_t      *sqld_verbose_types;      /* col, DEPRECATED, MS SQL
                                        Extension for dates */
dh_u32_t      sqld_v0;                  /* opaque */
dh_u32_t      sqld_v1;                  /* opaque */
dh_u32_t      sqld_v2;                  /* opaque */
dh_u32_t      sqld_v3;                  /* opaque */
dh_u32_t      sqld_v7;                  /* opaque */
dh_u32_t      sqld_v8;                  /* opaque */
```

Descriptions of the fields in the SQLDA follow:

SQLD_SIZE

Maximum number of variables (columns) that this SQLDA object can handle.

SQLD_NVARS

Number of variables (columns) found by a DESCRIBE operation. If you do not do a DESCRIBE operation, you **will** need to fill this in yourself.

SQLD_VARPTRS

Array of pointers to host variable names. The application sets elements of this array to point to the allocated buffer for storing the values retrieved in a FETCH operation. The buffer that array element SQLD_VARPTRS [i] points to must be of size (SQLD_LENGTHS [i] * SQLD_ARRAYSZ).

SQLD_LENGTHS

- CHARACTER — maximum size of the array. This needs to include a space for the trailing null(0) terminator character (+1 char).

- VARCHAR — maximum size of the array. This needs to include space for the trailing null(0) terminator character (+1 char).
- TIMESTAMP — sizeof (*tpe_timestamp_t*)
- BINARY — sizeof (*tpe_binary_t*)
- VARBINARY — sizeof (*tpe_binary_t*)
- LONG VARBINARY — sizeof (*tpe_bloblocator_t*), not supportedin ESQL/C
- BIGINT — sizeof (*tpe_bigint_t*)
- TINYINT — 1 byte
- BIT — 1 byte (only 1 bit used)
- INTEGER — 4 bytes
- SMALLINT — 2 bytes
- NUMERIC — sizeof (*tpe_num_t*)
- FLOAT— double precision
- REAL — single precision
- DATE — sizeof (*tpe_date_t*)
- TIME — sizeof (*tpe_time_t*)

SQLD_TYPES

Array of data types buffers.

SQLD_PRECISION

Array of precision values. These are the widths needed to display the data in characters without any special formatting. For example, an INTEGER has a PRECISION of 10. If the data is character, it does not include the trailing null(0) terminator character.

SQLD_SCALE

Array of scale values. These are the number of digits to the right of the decimal point for numeric data. It also applies to TIME and TIMESTAMP data where it is the number of fractional seconds.

SQLD_FLAGS

Array of boolean attributes. It indicates whether or not a variable or column can contain NULL values.

SQLD_IVARPTRS

Array of pointers to indicator variables. The application is responsible for assigning values in the array of pointers. If you do not wish to use indicator variables, you do not have to assign any values. If you do not assign values and they are needed by the server, an error will be generated. The indicator variable buffer that array element SQLD_IVARPTRS [i] points to must be of (sizeof (dh_i32_t) * SQLD_ARRAYSZ).

The value pointed to can take on three types of values. A -1 means that the particular value is NULL. A 0 (zero) menas that the value is good, ordinary data. A positive value indicates that your SQLD_LENGTHS value is not large enough to hold the result. In this case, the positive value is the size that was needed.

SQLD_VARNMSIZE

Maximum size of variable names.

SQLD_INDVARNMSZ

This attribute is unused and will be removed in a future release.

SQLD_VARNAMES

Array pointers to host variable names.

SQLD_INDVARNAMES

This attribute is unused and will be removed in a future release.

SQLD_ARRAYSZ

Size of an array the application allocates. Before fetching multiple rows in a single execution of a FETCH statement, the application sets this value to the number of rows to be fetched. Use this value in the computation of the size of storage arrays the application must allocate to store fetched values and to hold indicator variables.

10.6.3 Guidelines for Using an SQLDA

This section provides an outline for using an SQLDA. More information on each step is provided in following sections.

- Allocate an SQLDA by calling the PRO_SQLDA_Allocate function. Do not use a static or stack declaration. The function takes two arguments. The first indicates the maximum number of variables or columns that you expect to handle with this SQLDA object. The second states what the maximum number of bytes is that you wish to reserve for variable or column names when you are fetching data. For input SQLDA's, the second argument will generally be zero. The server will add 1 to your second argument to take into account the null(0) terminator byte on the name.

- You can reuse an SQLDA structure as long as multiple dynamic SQL statements using the descriptor area are not active at the same time, and the maximum SQLDA size is adequate for every SQL statement using the structure.

- Free an SQLDA by calling the PRO_SQLDA_Deallocate function. This function requires a parameter that is a pointer to the SQLDA structure. Remember that the client application is responsible for returning any memory allocated for the SQLD_VARPTRS and SQLD_IVARPTRS attributes.

- Your ESQLC application must allocate memory for storing the data values returned by the SQL statement and for holding the indicator variables. The application assigns values to the elements of the pointer arrays SQLD_VARPTRS and SQLD_IVARPTRS.

- If you are fetching multiple tuples (rows) in the execution of a single FETCH statement, the application must set the SQLD_ARRAYSZ correctly. The FETCH operation requires that SQLD_VARPTRS [i] points to a buffer of size (SQLD_LENGTHS [i] * SQLD_ARRAYSZ). Assign a null pointer to SQLD_IVARPTRS [i] if the statement does not require indicator variables. If the statement does require indicator variables, set SQLD_IVARPTRS [i] to point to a storage buffer for indicator variables. The indicator variables (columns) buffer size must be (sizeof (dh_i32_t) * SQLD_ARRAYSZ).

10.6.4 Allocating an SQLDA

Allocate an SQLDA structure by calling the *sqld_alloc( )* function. Do not use a static declaration. You must also declare a pointer to the SQLDA.

EXAMPLES

The first example shows a valid declaration:

```
struct sqlda *sqldaptr=(struct sqlda *)0;
   .
   .
   .
/* This example correctly calls PRO_SQLDA_Allocate */
sqldaptr = PRO_SQLDA_Allocate ( ... ) ;
   .
   .
   .
```

The next example shows an **invalid declaration** of an SQLDA structure:

```
/*
** THIS DECLARATION IS NOT CORRECT
*/
struct sqlda sqlda1 ;
```

The *sqld_alloc* function accepts a size argument that specifies the maximum number of variables for which the structure should be allocated. The syntax for the *sqld_alloc* function is:

SYNTAX

```
#include "sql_lib.h"
struct sqlda *PRO_SQLDA_Alocate(numVars, varNameLen)
dh_u32_t  numVars ;
dh_i16_t   varNameLen ;
```

The *sqld_alloc* function allocates an SQLDA from heap storage. The *numVars* variable specifies the maximum number of variables for which the SQLDA is allocated. *varNameLen* is the maximum number of characters to hold the names of the columns in the result table corresponding to the SQL statement. If your application does not need column names, set *varNameLen* to zero (0).

EXAMPLE

The following code fragment illustrates how to allocate an SQLDA and verify that the allocation succeeded:

```
#define  MAXVARS   20
#define  MAXVARNAMELEN  20

/* declare the pointer to an SQLDA */
struct sqlda *sqldaptr = (struct sqlda *)0 ;
   .
   .
   .
if (!(sqldaptr = PRO_SQLDA_Allocate(MAXVARS, MAXVARNAMELEN)))
     {
          printf ("sqld_alloc() returned an error \n") ;
          exit (1) ;
     }
printf ("sqld_alloc() allocation successful \n") ;
```

NOTE: The sqld_alloc function does not allocate memory for data buffers and indicator variable buffers. Allocation for data buffers and indicator variables must be performed by the application separately, and pointers to these must be set in the SQLDA.

Setting the SQLDA Types and Lengths

You use an SQLDA for both input host variables and output host variables. These variables are values that are not known beforehand. When you use an SQLDA to supply values to input host variables referenced in a dynamic non-SELECT statement, be sure to set the type and length of the values in the SQLDA. The SQLDA components SQLD_LENGTHS and SQLD_TYPES must be set correctly. For dynamic SELECT statements, issue a DESCRIBE statement to determine the types and lengths of the values the SELECT statement returns. When using an input SQLDA, you may also need to set the SQLD_SCALE and SQLD_PRECISION values for certain columns (e.g. set scale when the data type is NUMERIC). Also remember to add space in the SQLD_LENGTHS for the null(0) terminator byte when the data is CHARACTER or VARCHAR. If you do not do a DESCRIBE, you will need to set SQLD_NVARS yourself.

10.6.5 Using an SQLDA for Array Fetches

ESQLC supports fetches for multiple rows with the execution of a single FETCH statement. See the "Array Fetches" section in Chapter 7, "Query Statements," for a discussion of array fetches using explicitly declared arrays.

A similar technique is available in ESQLC using an SQLDA and dynamically allocated storage arrays. When a single execution of a FETCH operation is to retrieve multiple rows, the

application must set the SQLDA field SQLD_ARRAYSZ to the maximum number of rows that can be fetched. The application determines the size requirements for the values with a DESCRIBE operation, then allocates a buffer for storing the values the query is to retrieve, and a buffer for storing indicator variables, if required. The application sets the SQLDA components SQLD_VARPTRS array and SQLD_IVARPTRS to the correct sizes.

These are the general steps to follow when you are using dynamically declared arrays for array fetches:

1. PREPARE the SELECT statement.
2. DECLARE a cursor for the SELECT statement.
3. Allocate an INPUT SQLDA, if needed, for INPUT host variables.
4. Allocate an initial OUTPUT SQLDA.
5. DESCRIBE BIND VARIABLES, if needed, into the INPUT SQLDA.
6. OPEN CURSOR for the SELECT statement.
7. DESCRIBE SELECT LIST into the OUTPUT SQLDA.
8. Reallocate the OUTPUT SQLDA if SQLD_NVARS indicates the size is insufficient.
9. Set SQLD_ARRAYSZ to the maximum number of rows to fetch.
10. Assign values to the SQLDA components SQLD_LENGTHS and SQLD_TYPES with information from the DESCRIBE SELECT LIST.
11. Calculate the sizes needed for the output values array and indicator variables array.
12. Allocate storage for output values and indicator variables and store addresses in the OUTPUT SQLDA.
13. FETCH data recursively.
14. CLOSE CURSOR.
15. Free allocated storage for variables and indicator variables.
16. Free allocated INPUT SQLDA and OUTPUT SQLDA.

Allocating for SQLDA Data Buffers and Indicator Variables

The PRO_SQLDA_Allocate function does not allocate memory for data buffers or indicator variable buffers. The application program performs these allocations and sets pointers in the SQLDA to point to them.

Assign values to the SQLDA components SQLD_LENGTHS, SQLD_TYPES, and SQLD_ARRAYSZ before allocating data buffers and indicator variable buffers.

EXAMPLE

The following code fragment shows how to determine the size requirements for the buffers, assign the values in the SQLDA, and allocate the buffers:

```
/*
** Allocate for sqld_varptrs buffer and sqld_ivarptrs buffer;
** set pointers in the SQLDA to address the allocation.
*/

for (colindex = 0 ; colindex < sqldaptr->sqld_size ; colindex++)
{
     if (!( sqldaptr->sqld_varptrs [colindex] =
          (char *) calloc (sqldaptr->sqld_lengths [colindex],
                           sizeof (char) ))

     || !( sqldaptr->sqld_ivarptrs [colindex] = calloc(sizeof(dh_i32_t)
                           sizeof (char))))
     {
          printf ("Insufficient memory for dynamic buffer allocation \n") ;
          exit (1) ;
     }
}
```

NOTE: This example assumes that SQLD_ARRAYSZ is 1.

10.6.6 Freeing an SQLDA

Free a previously allocated SQLDA by using the *PRO_SQLDA_Deallocate* function call. The syntax for *PRO_SQLDA_Deallocate( )* is:

SYNTAX

```
#include "sqld_lib.h"
void PRO_SQLDA_Deallocate (sqldaptr)
struct sqlda *sqldaptr ;
```

EXAMPLE

The following example shows how to free an SQLDA:

```
PRO_SQLDA_Deallocate (sqldaptr) ;
```

You must free data buffers and indicator variable buffers before you call the *PRO_SQLDA_Deallocate* function.

CAUTION: Passing an invalid argument to PRO_SQLDA_Deallocate( ) or freeing an already freed SQLDA could cause abnormal termination of your application. Results are unpredictable.

10.7 DESCRIBE Statement

Use the DESCRIBE statement to obtain information about a prepared statement. This is an executable statement that can only be embedded in an ESQL program and cannot be dynamically prepared. The DESCRIBE statement can be used to describe INPUT host variables or OUTPUT host variables. This is the syntax for a DESCRIBE statement:

SYNTAX

```
EXEC SQL
  DESCRIBE BIND VARIABLES FOR statement_name INTO input_sqlda_name;

EXEC SQL
  DESCRIBE SELECT LIST FOR statement_name INTO output_sqlda_name;
```

Execute the DESCRIBE BIND VARIABLES statement after preparing an SQL statement and before executing the OPEN statement for the cursor. Specify the input SQLDA in the OPEN cursor statement with the USING DESCRIPTOR clause.

This is the syntax for a USING DESCRIPTOR clause in an OPEN cursor statement:

SYNTAX

```
EXEC SQL
  OPEN cursor_name USING DESCRIPTOR input_sqlda_name ;
```

EXAMPLE

The following DESCRIBE statement populates an SQLDA with information from a prepared SQL statement:

```
EXEC SQL BEGIN DECLARE SECTION ;
     CHAR  stmt_str[256] ;
EXEC SQL END DECLARE SECTION ;

struct sqlda *sqldaptr ;
   .
   .
   .
EXEC SQL PREPARE sel_stmt FROM :stmt_str ;
EXEC SQL DECLARE sel_cur CURSOR FOR sel_stmt ;
EXEC SQL OPEN sel_cur ;

/*
** Allocate an SQLDA
*/
if (!(sqldaptr = PRO_SQLDA_Allocate(20, 30)))
{
     printf ("Insufficient memory for SQLDA allocation\n") ;
     exit (1) ;
}

EXEC SQL
     DESCRIBE SELECT LIST FOR sel_stmt INTO sqldaptr ;
```

The sel_stmt is the output of a PREPARE operation on an SQL statement. When the DESCRIBE statement is executed, values are assigned to the variables in the SQLDA as follows:

- SQLD_NVARS contains the number of outputs returned. If the size of the sqlda passed is not enough to return information about all the result columns of the SELECT statement, SQLD_NVARS is negative. In this case, the absolute value of SQLD_NVARS gives the actual number of result columns of the SELECT statement. The application can use this to de-allocate the SQLDA, allocate a new SQLDA for the required size, and then execute the DESCRIBE statement again.
- SQLD_TYPES contains the data types of each column specified in the SELECT list.
- SQLD_LENGTHS contains the lengths of each column depending on the data type of the column.

10.8 SQLDA for Input Variables

Use an INPUT SQLDA to store the values for input host variables required in an SQL statement. Use a DESCRIBE BIND statement for an input SQLDA.

This is the syntax for a DESCRIBE BIND VARIABLES statement for an INPUT operation:

SYNTAX

```
EXEC SQL
     DESCRIBE BIND VARIABLES FOR statement_name INTO input_sqlda_name ;
```

The DESCRIBE BIND statement must be executed after the PREPARE step and before the OPEN operation.

EXAMPLE

The following example shows how to use a DESCRIBE BIND statement to determine the number of input host variable references in an SQL statement:

```
struct sqlda *isqlda ;

EXEC SQL BEGIN DECLARE SECTION ;
     char stmt_str [100] ;
EXEC SQL END DECLARE SECTION ;

strcpy (stmt_str,
     "insert into dept (deptno, dname, loc) values (:p1, :p2, :p3)") ;

EXEC SQL
     PREPARE ins_stmt FROM :stmt_str ;
EXEC SQL
     DECLARE ins_cur CURSOR FOR ins_stmt ;

/* Allocate for SQLDA */
isqlda = PRO_SQLDA_Allocate(20, 0) ;
if (!isqlda)
{
     printf ("Insufficient memory for SQLDA of size 20\n") ;
     exit (1) ;
}

/* Call DESCRIBE to get number of host variable references */
EXEC SQL
     DESCRIBE BIND VARIABLES FOR ins_stmt INTO isqlda ;

nvars = isqlda->sqld_nvars ;
if (nvars < 0) nvars = -nvars ;

/* Free SQLDA */
PRO_SQLDA_Deallocate(isqlda) ;

/* Allocate SQLDA for exact number of host variable references */
isqlda = PRO_SQLDA_Allocate(nvars, 0)
if (!isqlda)
{
     printf ("Insufficient memory for SQLDA \n") ;
     exit (1) ;
}
EXEC SQL
     OPEN ins_cur USING DESCRIPTOR isqlda ;
```

In this example, an SQLDA is first allocated for 20 variables. After the DESCRIBE is performed on the prepared statement, the SQLDA is once again allocated for the exact number of host variable references. Note that the second argument for *PRO_SQLDA_Allocate( )* is zero when allocating for an INPUT SQLDA.

10.9 SQLDA for Output Variables

For a dynamic SQL query, a DESCRIBE statement is required to associate items in the SELECT list with an output SQLDA. You use the DESCRIBE SELECT statement for the OUTPUT SQLDA. This is the syntax for a DESCRIBE SELECT LIST FOR statement:

SYNTAX

```
EXEC SQL
  DESCRIBE SELECT LIST FOR statement_name INTO output_sqlda_name ;
```

The DESCRIBE SELECT statement must occur after the PREPARE, DESCRIBE BIND, and OPEN statements, and before the FETCH statement.

EXAMPLE

The following example shows how to use the DESCRIBE SELECT statement:

```
struct sqlda *osqlda = (struct sqlda *)0 ;
strcpy (stmt, "select qty, product from orders where order_no < :p1") ;
EXEC SQL PREPARE stmtid FROM :stmt ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
for (order_no_v = 1002 ; order_no_v <= 1004 ; order_no_v++)
{
     EXEC SQL OPEN dyncur USING :order_no_v ;
     /* maxvars = 2, varnmsz = 20 */
     if (!(osqlda = PRO_SQLDA_Allocate(2, 20)))
     {
           printf ("PRO_SQLDA_Allocate returned err\n") ;
          goto err ;
     }
     for (;;)
     {
          short  nvars ;
          EXEC SQL DESCRIBE SELECT LIST FOR stmtid INTO osqlda ;
          if ((nvars = osqlda->sqld_nvars) < 0)
          {
               PRO_SQLDA_Deallocate(osqlda) ;
               if (!(osqlda = PRO_SQLDA_Allocate(-(nvars), 20)))
               {
                    printf ("PRO_SQLDA_Allocate returned err\n") ;
                    goto err ;
               }
               continue ;
          }
          break ;
     }
}
```

In this example, the SQLDA is allocated again when the size (first argument) provided in the *PRO_SQLDA_Allocate* function is insufficient. In this case the SQLDA component SQLD_NVARS contains a negative number equal to the negative of the actual number of outputs.

11

Transaction Management in ESQL-92

This chapter describes transaction management in ESQL. Specifically, it explains how to:

- Start a transaction
- End a transaction
- Set transaction isolation level
- Manage row and table locking in a transaction

11.1 Introduction to Transactions

A *transaction* is a sequence of operations on a database. Transactions are both *atomic* and *durable*. *Atomicity* refers to the property that either all the operations in a transaction are completed if the transaction is committed, or none of the operations are completed if the transaction is rolled back. *Durability* refers to the property that once you COMMIT a transaction, the changes made by the transaction are permanent.

Any executable statement is executed as part of a transaction. In an ESQL program, when you execute a statement and there is already an active transaction, the statement executes as part of the existing transaction. If an active transaction does not exist, execution of the SQL statement starts a new transaction. This transaction becomes the active transaction for that ESQL program. All subsequent SQL statements will execute as part of this transaction until the application ends the transaction with an explicit COMMIT or ROLLBACK operation.

11.1.1 Starting a Transaction

The database implicitly starts a transaction with the first executable SQL statement in an ESQL program. When an active transaction exists, a subsequent executable SQL statement does not begin a new transaction.

EXAMPLE

In the following code fragment, a new transaction begins when the database executes the CREATE TABLE statement. Executing the INSERT statement does not start a new transaction since an active transaction already exists:

```
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** CREATE the customer table in the database.
** This is the first executable statement, and
** the database implicitly starts a new transaction.
*/
EXEC SQL
     CREATE TABLE customer
     (
          cust_no INTEGER NOT NULL,
          last_name CHAR (30),
          street CHAR (30),
          city CHAR (20),
          state CHAR (2)
     ) ;

/*
** INSERT values into the customer table
*/
EXEC SQL
     INSERT INTO customer
          (cust_no, last_name, street, city, state)
     VALUES
          (:cust_no_v, :name_v, :street_v, :city_v, :state_v) ;

EXEC SQL
     COMMIT WORK ;

EXEC SQL
     DISCONNECT DEFAULT ;
```

NOTE: Declarative statements do not start a transaction.

11.1.2 COMMIT Statement

You can terminate a transaction in an application by using the COMMIT statement. When a COMMIT statement is executed, all the changes made to the database by the transaction are made permanent.

Depending on the isolation level of the transaction, changes made by one transaction might not be visible to other transactions before the transaction is committed. See the "Transaction Isolation Levels" section for more information. The default behavior is that database changes made by one transaction are not visible to any other transaction until you COMMIT the transaction. Once you COMMIT a transaction, the changes you made cannot be canceled.

EXAMPLE

The following example shows a COMMIT statement:

```
/*
** Explicitly end the transaction
*/
EXEC SQL COMMIT WORK ;
```

The COMMIT statement does not affect the contents of the host variables or the flow of control in the ESQL program.

11.1.3 ROLLBACK Statement

The ROLLBACK statement undoes all the changes made to the database within a transaction.

EXAMPLE

The following example shows how to use the ROLLBACK statement:

```
/*
** Undo transaction changes
*/
EXEC SQL ROLLBACK WORK ;
```

The ROLLBACK statement does not affect the contents of any host variables or the flow of control in the ESQL program. The database issues a ROLLBACK operation automatically when there is an abnormal termination of the application program. Often, the ROLLBACK statement is used in exception handler routines.

11.2 Transaction Isolation Levels

The degree to which one transaction can interfere with other transactions by accessing the same rows concurrently is determined by setting the *transaction isolation level* in the ESQL program.

Progress Software recommends that you specify the transaction isolation level number by *isolation_level_name*.

This is the syntax for the SET TRANSACTION ISOLATION LEVEL statement:

SYNTAX

```
SET TRANSACTION ISOLATION LEVEL isolation_level_name ;
```

isolation_level_name

SYNTAX

```
READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE
```

The ANSI/ISO SQL standard defines isolation levels in terms of the inconsistencies they allow:

`READ UNCOMMITTED`

Allows dirty reads, nonrepeatable reads, and phantoms. When a record is read, no record locks are aquired. This forces read-only use. Allows a user application to read records that were modified by other applications but have not yet been committed.

`READ COMMITTED`

Prohibits dirty reads; allows nonrepeatable reads and phantoms. Whenever a record is read, a share lock is acquired on that record. The duration of the lock varies. Disallows the reading of uncommitted modified records. However, if a record is read multiple times in the same transaction, the state of the record remains the same.

`REPEATABLE READ`

Prohibits dirty reads and nonrepeatable reads; allows phantoms. Whenever a record is read, a share lock is acquired on that record and held until the end of the current transaction. Disallows the reading of uncommitted modified records. If a record is read multiple times in the same transaction, the state of the record remains the same.

`REPEATABLE READ` is the default isolation level.

`SERIALIZABLE`

Prohibits dirty reads, nonrepeatable reads, and phantoms.If an application executes the same `SELECT` statement more than once within the same transaction, the same set of rows is retrieved every time. Guarantees that concurrent transactions will not affect each other, and that they will behave as if they were executing serially, not concurrently.

Whenever a table is accessed, the entire table is locked with an appropriate lock. The talbe lock is held until the end of the current transaction.

NOTES

- The isolation level *`SERIALIZABLE`* guarantees the highest consistency.
- The isolation level *`READ UNCOMMITTED`* guarantees the least consistency.
- The default isolation level is *`REPEATABLE READ`*, which prohibits non-repeatable read operations.

For more information on setting isolation levels in transactions, see SET TRANSACTION ISOLATION LEVEL in Chapter 3, “SQL-92 Statements,” of the *Progress SQL-92 Guide and Reference*.

EXAMPLE

The following code fragment illustrates how to execute a query at the READ COMMITTED isolation level:

```
EXEC SQL CONNECT TO DEFAULT ;

EXEC SQL WHENEVER SQLERROR GOTO do_rollback ;

EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED ;

EXEC SQL
     SELECT last_name, city
     INTO :cust_no_v, :name_v
     FROM customer
     WHERE cust_no = 1024 ;
   .
   .
   .

EXEC SQL COMMIT WORK ;
EXEC SQL DISCONNECT DEFAULT ;
exit (0) ;

do_rollback:
     if (sqlca.sqlcode < 0)
     {
          strncpy (errmesg, sqlca.sqlerrm, sqlca.sqlerrml);
          errmesg [sqlca.sqlerrml] = '\0' ;
          printf ("Error : %s\n", errmesg);
     }

EXEC SQL WHENEVER SQLERROR CONTINUE ;
EXEC SQL ROLLBACK WORK ;
EXEC SQL DISCONNECT DEFAULT ;
exit (1) ;
```

11.3 Locking and Transactions

This section discusses general points about programs and *row locking*. For more information on *table locking* and the LOCK TABLE statement, see Chapter 3, "SQL-92 Statements," in the *Progress SQL-92 Guide and Reference*.

During execution of an ESQL program, transactions implicitly lock rows in a database. This is done so that the data in the database remains consistent. The implicit locks are released when you commit the transaction or disconnect from the database.

Typically, a storage system supports two types of locks:

- *SHARE locks*

  The database acquires SHARE locks on rows that a transaction has read. A SHARE lock allows other transactions to read the row but prevents others from modifying the row until this transaction issues a COMMIT or a ROLLBACK, or disconnects from the database.

- *EXCLUSIVE locks*

  The database acquires EXCLUSIVE locks on rows that have been modified by a transaction. EXCLUSIVE locks prevent other transactions from either reading or modifying the rows until this transaction issues either a COMMIT or a ROLLBACK, or disconnects from the database.

In applications where a large number of rows will be accessed for either reading or modifying, ESQL provides an explicit locking construct for locking all the rows of a table. The *LOCK TABLE* statement explicitly locks a table in either SHARE or EXCLUSIVE mode.

EXAMPLE

The following example shows acquiring a lock in the EXCLUSIVE mode for the customer table:

```
EXEC SQL
     LOCK TABLE customer IN EXCLUSIVE MODE ;
```

This statement prevents other transactions from either reading or modifying the table customer until the transaction issues a COMMIT or ROLLBACK, or disconnects from the database.

EXAMPLE

The following example shows how to acquire a lock in SHARE mode for the orders table:

```
EXEC SQL
     LOCK TABLE orders IN SHARE MODE ;
```

This statement prevents other transactions from modifying the orders table until the transaction issues either a COMMIT or a ROLLBACK. You can use explicit locking to improve the performance of a single transaction at the cost of decreasing the concurrency of the system, and potentially blocking other transactions. The increased performance comes from reducing the overhead imposed by the implicit locking mechanism, along with eliminating any potential waits for acquiring row level locks for the table.

11.4 Abnormal Termination of an ESQL Application Program

If an active transaction exists when an ESQL application disconnects from a database, then the database rolls back the transaction automatically. This behavior is similar to when an ESQL application terminates abnormally. In this case, disconnecting from the database is also automatic.

11.4.1 Forced Rollback of a Transaction

The database issues a forced rollback of an active transaction when an ESQL program disconnects from the database or makes an abnormal exit. This forced rollback keeps the database consistent.

When some serious errors occur, a transaction might be implicitly marked for ROLLBACK. This can be detected by the information in the SQLCA. If a transaction is marked for ROLLBACK, no SQL statement can be executed on behalf of that transaction. In such a case, the application must roll back the current transaction before proceeding with execution of the next SQL statement.

11.4.2 Interrupting the Execution of an SQL Statement

During the execution of an SQL statement an interrupt can occur. When this happens the statement being executed fails and the database returns an error code in the SQLCA. The database rolls back any changes made by this particular statement. The database returns to the state it held before execution of the SQL statement.

EXAMPLE

In this sequence of operations, if the interrupt occurs during the execution of the UPDATE statement, then the UPDATE statement returns an error and all the changes made by the UPDATE statement are undone. The changes made by the INSERT statement are retained. If the statement execution proceeds to the COMMIT statement, then only the INSERT operation is made permanent, as shown in the following example:

```
SQL CONNECT TO DEFAULT ;
   .
   .
   .

EXEC SQL
     INSERT INTO orders
     VALUES (:no_v, :date_v, :prod_v, :qty_v) ;

/*
** If execution of the UPDATE statement is interrupted,
** the database rolls back any database modifications
** completed by the UPDATE statement before the interrupt.
**
** Database modifications completed by the
** INSERT statement are NOT undone.
**
** The application program should evaluate the success of each
** operation and take appropriate action.
*/
EXEC SQL
     UPDATE orders
     SET product = :prod_x
     WHERE order_date > :ord_date ;

/* End Transaction */
EXEC SQL COMMIT WORK ;
EXEC SQL DISCONNECT DEFAULT ;
```

12

Guidelines for Developing ESQL-92 Applications

This chapter contains general guidelines for developing ESQL-92 applications. Specifically, it illustrates these operations available in Progress SQL-92:

- Computation
- Condition evaluation
- Using indicator variables
- Using scalar functions
- Processing static statements
- Processing dynamic statements

12.1 Coding Guidelines

Some general coding guidelines for an application developer using ESQL follow:

- The character strings used in an ESQL program must be null terminated.
- The SQL-92 operator that shows equality is a single equals sign (=). The C Language operator that evaluates for equality is a double equals sign (==), and in C the single equals sign (=) is the assignment operator. Often all of these operators will be in an embedded SQL C Language source program.
- Use the WHENEVER statement in an ESQL program to check for error conditions. This statement provides more flexibility and reduces the code size of the application.
- Problems occur if any of the SQL reserved words are in #define statements of included header files and you specify the +P option when executing ESQLC. When you specify the +P option, the source files are passed through the C preprocessor before the ESQLC statements are translated. One frequent problem is the reserved word NULL that the standard header file `stdio.h` #defines. In such cases, use one of the following solutions:
  - Avoid using the +P option when feasible.
  - Avoid including the header file that contains the definition of the reserved word.
  - Reverse the case of a reserved word such as NULL when you use it in an ESQL statement.

EXAMPLE

The following example reverses the case of the reserved word NULL:

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

This does not affect ESQLC since reserved words in ESQL statements are not case sensitive.

- Use the owner name of the table when referencing tables in an ESQL program instead of using a nonqualified table name.

EXAMPLE

For example, to access the customer table, specify the owner name john with the customer table:

```
EXEC SQL
     SELECT cust_no, last_name
     FROM john.customer ;
```

- When you declare a host variable array, do not use an expression in the array subscript.

EXAMPLES

For example, the following declaration in the declare section is invalid, and the precompiler returns an error:

```
#define MAXNAMELEN 18
/*
** THIS DECLARATION IS INVALID
*/
EXEC SQL BEGIN DECLARE SECTION ;
     char last_name [MAXNAMELEN + 1] ;
EXEC SQL END DECLARE SECTION ;
```

MAXNAMELEN is defined with a #define symbol. To handle such requirements, define a new symbol for the required value and use the new symbol, as shown in this example:

```
#define MAXNAMELEN_P1 19

EXEC SQL BEGIN DECLARE SECTION ;
  char last_name [MAXNAMELEN_P1] ;
EXEC SQL END DECLARE SECTION ;
```

- You can use #define symbols in SQL statements wherever you can use constants. If your source program contains #define statements, you must use the +P option when you invoke ESQLC.

EXAMPLE

The following example shows how to use the #define symbol:

```
MAX_SALARY:
     #define MAX_SALARY 10000
EXEC SQL
     SELECT ename, salary
     FROM employee
     WHERE salary = MAX_SALARY ;
```

Specify all the columns in the SELECT list of a SELECT statement instead of specifying a (*). This might improve the readability of the SELECT statement. In addition, if more columns are added to the database table definition later, you do not have to change your statement unless you choose to select the additional columns.

12.2 Using SQL for Computation

You can use an SQL SELECT statement to perform computations in an ESQL program by using the *syscalctable* system table for computations. This system table has owner sysprogress and contains exactly one row with a single column.

EXAMPLE

The following example illustrates using the *syscalctable* system table for computations.

```
EXEC SQL
     SELECT expression INTO :host_var FROM sysprogress.syscalctable ;
```

NOTE: For a computation operation you can use any table that contains exactly one row. *sysprogress.syscalctable* is a table with exactly one row.

The expression in the SELECT list can contain constants or host variables and can include scalar functions.

EXAMPLE

The following examples show some of the ways you can use the SELECT statement to perform computations:

```
EXEC SQL BEGIN DECLARE SECTION ;
     long result ;
     long num ;
     long val ;
     DATE dtval ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .

EXEC SQL
     SELECT (100 * 12 (1234 - 354))
     INTO :result
     FROM sysprogress.syscalctable ;


EXEC SQL
     SELECT (:num * 10 + ABS ((360 - :val)))
     INTO :result
     FROM sysprogress.syscalctable ;


EXEC SQL
     SELECT NEXT_DAY (TO_DATE ("8/10/1998"), "WEDNESDAY")
     INTO :dtval
     FROM sysprogress.syscalctable ;
```

Using the *syscalctable* system table provides flexibility and eases the evaluation of expressions, especially when the expressions involve values for NUMERIC, DATE, or TIME. See Appendix B, “Progress SQL-92 System Catalog Tables,” in the *Progress SQL-92 Guide and Reference* for information on the *syscalctable* system table and all system tables.

12.3 Using SQL for Condition Evaluation

You can use an SQL SELECT statement for condition evaluation in an ESQL program. The condition evaluation is performed using the *syscalctable* system table.

EXAMPLE

The following example shows how to use the SELECT statement to check whether the value in the ship_date column is valid:

```
EXEC SQL
     SELECT 1
     INTO :result
     FROM sysprogress.syscalctable
     WHERE :ship_date > SYSDATE
     AND :ship_date < ADD_MONTHS (:order_date, 1) ;

if (sqlca.sqlcode == 0)
     printf ("ship_date valid \n") ;
else
if (sqlca.sqlcode == SQL_NOT_FOUND)
     printf ("ship_date invalid \n") ;
else
     printf ("Error \n") ;
```

The above example returns zero (0) in SQLCA.SQLCODE if the validation is successful. If the validation is unsuccessful SQLCODE is not set to zero.

12.4 Using Indicator Variables

Indicator variables can be used with input host variables and output host variables. A NULL value can be inserted when an indicator variable is used along with the input host variable, if the indicator variable is set to -1. In the output variables, indicator variables are used to check for NULL values and other error conditions.

When you issue a query, use an indicator variable to check for NULL values on associated columns.

Table 12–1 lists the indicator variable values and their meanings.

Table 12–1: Indicator Variable Values and Their Meanings

Value	Meaning of Indicator Variable Value
0	The associated host variable contains a non-NULL value. If set by SQL in a fetch operation, 0 also indicates that the value in the host variable has not been truncated.
- 1	The returned value in an OUTPUT operation is NULL and there is no value in the associated host variable. The value of the associated host variable for an UPDATE or INSERT operation is NULL.
> 0	The host variable size was too small to contain the returned value in an OUTPUT operation, and the returned value in the host variable has been truncated. The indicator variable itself contains the > 0 length of the returned value before it was truncated.

12.5 Using Scalar Functions

Scalar functions accept as arguments a collection of values derived from one or more columns corresponding to one row, from either a database table or an intermediate result table. Scalar functions return one value.

EXAMPLES

The following examples illustrate scalar functions in SQL statements:

```
EXEC SQL
     SELECT order_no, product
     FROM orders
     WHERE ABS (qty - :old_qty) > 10000 ;

EXEC SQL
     SELECT order_no, product, qty
     FROM orders
     WHERE order_date = TO_DATE ('01/02/1993') ;

EXEC SQL
     SELECT MONTHS_BETWEEN (SYSDATE, order_date)
     FROM orders
     WHERE order_no = 1005 ;
```

The DECODE function accepts a column name, a column value, and its substitute value. Optionally, you can specify a default value. This example compares the deptno column to the department code, and returns the corresponding department name. If there is no match, the example returns the default string NOT ASSIGNED. The DECODE function is similar to the C Language *switch* statement.

```
EXEC SQL
     SELECT ename, DECODE
          (
          deptno,
          10, 'ACCOUNTS ',
          20, 'RESEARCH ',
          30, 'SALES ',
          40, 'SUPPORT ',
          'NOT ASSIGNED'
          )
     FROM employee ;
```

You can use the NVL function to check whether a column value is NULL. If NULL, you can specify a substitute value to return. This example returns the substitute zero value if the commission column contains a NULL value:

```
EXEC SQL
     SELECT salary + NVL (commission, 0)
     FROM employee ;
```

12.6 Using Static and Dynamic Statements

Since static SQL statements are simpler and easier to use than dynamic SQL statements, use static SQL statements wherever feasible.

13

Handling Data Types in ESQL-92

This chapter describes the data types that you can use in an ESQL application program and discusses data conversion and data comparison mechanisms.

13.1 Introduction

The database types supported in Progress SQL-92 have two representations:

- *Database storage representation*
- *Host language representation*

The database storage representation refers to the way a value is stored in the database. The host language representation refers to the way a value is represented in the C Language. Progress SQL-92 performs the necessary data conversions from one form to another. The application does not manage database storage representation.

See Chapter 2, "SQL-92 Language Elements," in the *Progress SQL-92 Guide and Reference* for additional information on supported data types.

The next sections discusses the different categories of data types supported by Progress SQL-92 and how you use them in ESQL programs as INPUT and OUTPUT host variables.

13.2 Character Data Types

This section describes the supported *CHARACTER* data types.

13.2.1 CHARACTER Data Type

The Progress SQL-92 data type *CHARACTER* (alias CHAR) corresponds to a NULL terminated character string with a maximum length of 2000 characters. The host language representation is a C Language character string.

You can use a literal character string or a host variable declared as a character array as an INPUT host variable. You must enclose string constants or literals in single or double quotes. You can also use a host variable declared as a character array as an OUTPUT host variable.

EXAMPLES

The SELECT statement in the following code fragment uses an INPUT host variable and an OUTPUT host variable, each with data type CHAR:

```
EXEC SQL BEGIN DECLARE SECTION ;
     char city_v[19] ;
     char name_v[19] ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
EXEC SQL
     SELECT city
     INTO :city_v
     FROM customer
     WHERE last_name = :name_v ;
```

The city_v variable is an OUTPUT host variable, and name_v is an INPUT host variable. Each is declared as a character array in the DECLARE SECTION. An INPUT host variable can also be a character literal, as shown in this example:

```
WHERE last_name = 'JOHN' ;
```

You can use the scalar functions SUBSTR, INSTR, LENGTH, LOWER, and UPPER to manipulate character strings. See Chapter 4, "SQL-92 Functions," in the *Progress SQL-92 Guide and Reference* for additional information on scalar functions.

13.2.2 VARCHAR Data Type

Progress SQL-92 supports *VARCHAR* as a database data type with a maximum length of 31995 characters. However, the C Language representation for the VARCHAR data type is a character string. If your application requires a column of VARCHAR type, declare it in your C program as VARCHAR.

NOTE: Other Progress SQL-92 tools support VARCHAR. Use the VARCHAR data type in ESQL applications.

13.3 Exact Numeric Data Types

This section describes the supported *EXACT NUMERIC* data types.

13.3.1 INTEGER Data Type

The *INTEGER* data type corresponds to an integer of 4 bytes. The database representation, as well as host language representation, is a long integer. You must express integer constants in decimal notation without embedded commas and without a decimal point.

The integer data can be preceded with an optional plus or minus sign. The range of values for a column of this type is -2147483648 to +2147483647.

EXAMPLE

The following example shows how to use the INTEGER data type expressed as a host language long integer:

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

EXEC SQL
     SELECT qty
     INTO :qty_v
     FROM orders
     WHERE order_no = 1002 ;
```

13.3.2 SMALLINT Data Type

The *SMALLINT* data type corresponds to an integer value of 2 bytes. The database representation, as well as the host language representation, is a signed short integer. The range of values corresponding to a column of this type is -32768 to +32767.

EXAMPLE

The following example shows how to declare and use a SMALLINT data type:

```
EXEC SQL BEGIN DECLARE SECTION ;
     short deptno_v ;
     char ename_v [19] ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
EXEC SQL
     SELECT ename, deptno
     INTO :ename_v, :deptno_v
     FROM employee
     WHERE empno = 2828 ;
```

13.3.3 TINYINT Data Type

The *TINYINT* data type corresponds to an integer value in the range -128 to 127 inclusive.

```
typedef struct {
     unsigned long l1 ;
     long h1;
} tpe_tinyint_t ;
```

EXAMPLE

The following example shows how to declare a TINYINT data type:

```
EXEC SQL BEGIN DECLARE SECTION ;
    TINYINT tiny_v ;
EXEC SQL END DECLARE SECTION ;
```

13.3.4 NUMERIC or DECIMAL Data Type

The *NUMERIC* data type, which is the same as the *DECIMAL* data type, corresponds to a number with a given precision and scale. The database representation and the host language representation for the NUMERIC type is the following C Language structure:

```
typedef struct {
     short dec_num ;
     char dec_digits[25] ;
} tpe_num_t ;
```

The **dec_num** field in the structure contains the number of valid bytes in the **dec_digits** array. The most significant bit of the first byte of **dec_digits[0]** contains the sign bit: 1 for positive and 0 for negative.

The least significant 7 bits of **dec_digits[0]** contain the exponent in excess-64 format. The rest of the bytes in the **dec_digits** array contain the base-100 digits of the number value, with two digits in each byte. You can use SQL-92 functions to manipulate the number values in this format.

The range of values of a NUMERIC type column is -n to n, where n is the largest number that can be represented with the specified precision and scale.

EXAMPLE

The following example shows how to use a NUMERIC type:

```
EXEC SQL BEGIN DECLARE SECTION ;
     numeric commission_v ;
     char ename_v[19] ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL
     SELECT ename, commission
     INTO :ename_v, :commission_v
     FROM employee
     WHERE empno = 2040 ;
```

In an application program, you can use the *tpe_num_t* structure to access the NUMERIC database type.

13.4 Approximate Numeric Data Types

This section describes the supported *APPROXIMATE NUMERIC* data types.

13.4.1 REAL Data Type

The *REAL* data type corresponds to a single-precision floating-point number. The database representation and the host language representation are the same as the C Language float type.

EXAMPLE

The following example shows how to use the REAL data type:

```
EXEC SQL BEGIN DECLARE SECTION ;
     float val ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL
     SELECT 120.00 + 100.50
     INTO :val
     FROM syscalctable ;
```

13.4.2 FLOAT Data Type

The *FLOAT* data type corresponds to a double-precision floating-point number. The database representation and the host language representation are the same as the C Language DOUBLE data type.

EXAMPLE

The following example shows how to use the FLOAT data type:

```
EXEC SQL BEGIN DECLARE SECTION ;
     double salary_v ;
     char ename_v[19] ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL
     SELECT ename, salary
     INTO :ename_v, :salary_v
     FROM employee
     WHERE empno = 2040 ;
```

13.4.3 DOUBLE PRECISION Data Type

The *DOUBLE PRECISION* data type corresponds to a double-precision floating-point number equivalent to the C Language double type. The database representation and the host language representation are the same as the C Language DOUBLE data type.

13.5 Date-time Data Types

This section describes the supported *DATE-TIME* data types.

13.5.1 DATE Data Type

The *DATE* data type corresponds to a date value with three parts: day-of-month, month, and year. The database representation is a long integer. The host language representation is equivalent to the following C Language structure:

```
typedef struct {
     short  year ;
     unsigned char month ;
     unsigned char day ;
} tpe_date_t ;
```

The year range is 1 to 9999. The month range is 1 to 12. The lower limit for the day is 1, and the upper limit depends on the month and the year.

EXAMPLES

The following examples show how to use the DATE data type:

```
EXEC SQL BEGIN DECLARE SECTION ;
DATE date_v ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL
     SELECT order_date
     INTO :date_v
     FROM orders
     WHERE cust_no = 2002 ;
```

You can use the DATE data type in a WHERE clause, as shown:

```
WHERE order_date = :date_v ;
```

Some of the scalar functions available for manipulating date values are ADD_MONTHS, MONTHS_BETWEEN, and TO_DATE. See Chapter 4, "SQL-92 Functions," in the *Progress SQL-92 Guide and Reference* for additional information on scalar functions.

13.5.2 TIME Data Type

The *TIME* data type corresponds to the time value with four parts: hours, minutes, seconds, and milliseconds. The database representation is a long integer. The host language representation is equivalent to the following C Language structure:

```
typedef struct {
     unsigned char hours ;
     unsigned char mins ;
     unsigned char secs ;
     unsigned short msecs ;
} tpe_time_t ;
```

The hours range is 0 to 23. The minutes range is 0 to 59. The range for the seconds part is 0 to 59. The milliseconds range is 0 to 999.

EXAMPLE

The following example shows how to use the TIME data type:

```
EXEC SQL BEGIN DECLARE SECTION ;
     TIME time_v ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL
     SELECT arrival_time
     INTO :time_v
     FROM arrivals
     WHERE train_no = 230 ;
```

Some of the scalar functions available for manipulating TIME data are TO_TIME, HOUR, MINUTE, SECOND, etc. See Chapter 4, "SQL-92 Functions," in the *Progress SQL-92 Guide and Reference* for additional information on scalar functions.

13.5.3 TIMESTAMP Data Type

The *TIMESTAMP* data type combines the parts of DATE and TIME. The database representation and the host language representation for the TIMESTAMP data type is the following C Language structure:

```
typedef struct {
     tpe_date_t ts_date ;
     tpe_time_t ts_time ;
} tpe_timestamp_t ;
```

13.6 Bit String Data Types

This section describes the supported Bit String data types.

13.6.1 BIT Data Type

The *BIT* data type corresponds to a single bit value of 0 or 1.

SQL statements can assign and compare values in BIT columns to and from columns of types CHAR, VARCHAR, BINARY, VARBINARY, TINYINT, SMALLINT, and INTEGER. However, in assignments from BINARY, and VARBINARY, the value of the first four bits must be 0001 or 0000.

No arithmetic operations are allowed on BIT columns.

13.6.2 BINARY Data Type

The *BINARY* data type corresponds to a bit field of the specified length of bytes. The default length is 1 byte. The maximum length is 2000 bytes.

BINARY data can be assigned and compared to and from columns of type BIT, CHAR, and VARBINARY types. Arithmetic operations are not allowed. The database representation and the host language representation for the BINARY data type is the following C Language structure:

```
typedef struct {
     long  tb_len ;
     unsigned char  tb_data[TPE_MAX_FLDLEN+1] ;
} tpe_binary_t ;
```

13.6.3 VARBINARY Data Type

The *VARBINARY* data type corresponds to a variable-length bit field with the maximum length specified. The default length is 1 byte and maximum length is 31960 bytes. Otherwise, VARBINARY columns have the same characteristics as BINARY.

13.7 Data Conversion

This section discusses the data conversions that are supported in ESQL.

13.7.1 Implicit Data Conversion

Progress SQL-92 performs the required data conversions between data types wherever possible. For example, when delivering each SELECT list value to the host variables in an INTO clause, Progress SQL-92 implicitly converts the data value, if necessary, from the database representation to the host language representation.

If the database column is of type SMALLINT, Progress SQL-92 retrieves the value of that column in to a host variable of type long, and automatically converts the value from SMALLINT to long integer type. Presently, implicit type conversions between all data types are **not supported**. For example, if you use an INTEGER host variable to hold a CHARACTER value, there is no implicit data conversion. If you do not explicitly convert the INTEGER variable to CHAR and you attempt to access it as CHAR, an error is returned.

13.7.2 Explicit Data Conversion Using *tpe_conv_data( )*

To convert data explicitly from one data type to another, use the *tpe_conv_data* SQL-92 API function. This is the syntax for the *tpe_conv_data* function:

SYNTAX

```
tpe_status_t
tpe_conv_data (dtype, buflen, buf, fmt, odtype, obuflen, obuf)

int  dtype;
int  buflen;
void *buf;
char *fmt;
int  odtype;
int  obuflen;
void *obuf;
```

dtype

Data type of the INPUT buffer being converted. See the list of supported defined data types.

buflen

The length of the INPUT buffer `buf`.

buf

A buffer containing the INPUT data and whose type is specified in `dtype`.

`fmt`

A format string that specifies the output format. Valid only if *dtype* is TPE_DT_DATE and *odtype* is TPE_DT_CHAR. See Chapter 2, "SQL Language Elements," in the *Progress SQL-92 Guide and Reference* for information on date-time format strings.

odtype

Data type of the OUTPUT required after conversion.

obuflen

The length of the OUTPUT buffer *obuf*.

obuf

The target OUTPUT buffer for the converted data.

dtype

Must be one of the data type symbolic names defined in `$DLC/include/sql_lib.h`. The defined symbolic names include:

- TPE_DT_CHAR
- TPE_DT_VARCHAR
- TPE_DT_INTEGER
- TPE_DT_SMALLINT
- TPE_DT_TINYINT
- TPE_DT_NUMERIC
- TPE_DT_FLOAT

- TPE_DT_REAL
- TPE_DT_DATE
- TPE_DT_TIME
- TPE_DT_TIMESTAMP
- TPE_DT_BIT
- TPE_DT_BINARY
- TPE_DT_VARBINARY

EXAMPLE

The following code fragment illustrates how to convert CHARACTER data to INTEGER data:

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

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

In this example **indata[ ]** is the buffer containing INPUT data and **outval** is the buffer where the converted data is returned.

The *tpe_conv_data* function **does** perform type conversions between character types and exact or approximate numeric types, unlike the limitation imposed by implicit type conversions.

EXAMPLE

The following code fragment illustrates how to convert NUMERIC Data to CHARACTER data:

```
EXEC SQL BEGIN DECLARE SECTION ;
     char  var0[52] ;
     numeric var7 ;
EXEC SQL END DECLARE SECTION ;


EXEC SQL
     SELECT fld2 INTO :var7 FROM compt WHERE fld1='10' ;
     tpe_conv_data (TPE_DT_NUMERIC, sizeof(tpe_num_t), &var7,
                    "", TPE_DTCHAR, 52, var0) ;
     printf("var0 = %s\n", var0 ) ;
```

This example illustrates using *tpe_conv_data( )* to convert NUMERIC data retrieved from the database into character data for use by the host program. In the example var7 is the buffer containing input data and var0 is the target for output data.

13.7.3 Explicit Data Conversion Using Scalar Functions

Alternatively, you can convert one data type to another data type using scalar functions. See Chapter 4, "SQL-92 Functions," in the *Progress SQL-92 Guide and Reference* for additional information on scalar functions.

EXAMPLE

The following examples show two of the conversions available using scalar functions:

```
EXEC SQL BEGIN DECLARE SECTION ;
     numeric num_val_v ;
     date date_val_v ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL
     SELECT TO_DATE ('10/10/1999')
     INTO :date_val_v
     FROM sysprogress.syscalctable ;

EXEC SQL
     SELECT TO_NUMBER ('1234')
     INTO :num_val_v
     FROM sysprogress.syscalctable ;
```

The first example converts the character expression '10/10/1999' to a DATE data type, using the TO_DATE scalar function. The result is returned in date_val_v. The second example converts character expression '1234' to NUMERIC type using the TO_NUMBER scalar function. The result is returned in num_val_v.

Some of the other scalar functions available for conversion are TO_CHAR and TO_TIME. See Chapter 4, "SQL-92 Functions," in the *Progress SQL-92 Guide and Reference* for additional information on scalar functions.

13.7.4 Explicit Data Conversion Using SQL and *syscalctable*

You can also convert from one data type to another without using scalar functions in SQL-92 statements. The following example shows how to convert an INTEGER value to a DOUBLE PRECISION value, using the *syscalctable* system table:

```
EXEC SQL BEGIN DECLARE SECTION ;
     INTEGER qty_v ;
     DOUBLE double_val_v ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL
     SELECT :qty_v
     INTO :double_val_v
     FROM sysprogress.syscalctable ;
```

NOTE: When your application must be portable to another RDBMS, use SQL-92 statements for data type conversion in ESQL applications. When performance is the priority, use the SQL-92 API call *tpe_conv_data* to improve the performance of your application program.

13.8 Data Comparison

This section describes how to perform data comparisons. ESQL allows the comparison of two data values of the same database type.

13.8.1 Explicit Data Comparison Using *tpe_compare_data( )*

Use the SQL-92 API function *tpe_compare_data* to compare two data values of the same type. The syntax for *tpe_compare_data* is:

SYNTAX

```
tpe_status_t
tpe_compare_data (dtype, buf1len, buf1, buf2len, buf2, result)

int dtype;
int buf1len;
void *buf1;
int buf2len;
void *buf2;
int *result ;
```

dtype

Data type of the two values that are being compared.

buf1len

Length of the buffer containing the first value.

buf

A pointer to the buffer containing the first value.

buf2len

Length of the buffer containing the second value.

buf2

A pointer to the buffer containing the second value.

result

An integer that points to a location for returning the result. Table 13–1 shows the range of integer values returned by the *tpe_compare_data* function into *result*.

Table 13–1: Values Returned by *tpe_compare_data()*

Integer Value	Meaning of Integer Value Returned by *tpe_compare_data()*
1	First data value is greater than the second data value.
-1	First data value is less than the second data value.
0	The two data values are equal.

dtype

Must be one of the data type symbolic names defined in `$DLC/include/sql_lib.h`. The defined symbolic names include:

- TPE_DT_CHAR
- TPE_DT_VARCHAR
- TPE_DT_INTEGER
- TPE_DT_SMALLINT
- TPE_DT_TINYINT
- TPE_DT_NUMERIC
- TPE_DT_FLOAT
- TPE_DT_REAL
- TPE_DT_DATE
- TPE_DT_TIME

- TPE_DT_TIMESTAMP
- TPE_DT_BIT
- TPE_DT_BINARY
- TPE_DT_VARBINARY

The *tpe_compare_data* function returns a 0 if the compare operation succeeds. If the compare operation is not successful, the function returns a non-zero status code.

EXAMPLE

The following code fragment shows the comparison of two NUMERIC data elements. This example compares val1_v and val2_v and returns the result in the result_v variable:

```
EXEC SQL BEGIN DECLARE SECTION ;
     numeric val1_v ;
     numeric val2_v ;
     numeric result_v ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
status = tpe_compare_data (TPE_DT_NUMERIC, sizeof(tpe_num_t),
        (void *)&val1_v, sizeof(tpe_num_t), (void *)&val2_v,
        &result_v) ;
if (status == 0)
{
     if (result_v > 0)
          printf("val1_v > val2_v ") ;
     else
          if (result_v < 0)
               printf("val1_v < val2_v ") ;
          else
               printf("val1_v = val2_v ") ;
}
/*
** Error handling logic here, for unsuccessful tpe_compare_data operation.
*/
```

13.8.2 Explicit Data Comparison Using SQL and *syscalctable*

Alternatively, you can compare two data values using an SQL-92 statement, as follows:

```
EXEC SQL BEGIN DECLARE SECTION ;
    DATE date1_v ;
    DATE date2_v ;
    SHORT result_v ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
EXEC SQL
     SELECT :date1_v - :date2_v
     INTO :result_v
     FROM sysprogress.syscalctable ;
if (sqlca.sqlcode == 0)
{
     if (result_v > 0)
          printf ("date1_v > date2_v ") ;
     else
          if (result_v < 0)
               printf ("date1_v < date2_v ") ;
          else
               printf ("date1_v = date2_v ") ;
}
/*
** Error handling logic here, for unsuccessful SELECT operation.
*/
```

The result is the difference in the number of days between the two given dates, date1_v and date2_v.

NOTE: When your application must be portable to another RDBMS, use SQL-92 statements for data comparisons. When performance is the priority, use the SQL-92 API call *tpe_compare_data* to improve the performance of your application program.

14

Embedded SQL-92 Reference

This reference chapter contains information on statements used for processing embedded SQL-92 statements. For information on all supported SQL-92 statements, see the *Progress SQL-92 Guide and Reference*.

See Chapter 10, "Dynamic SQL Management in ESQL-92," for information on the combinations of steps required to process a statement using dynamic SQL-92.

All of the variants of language elements and SQL-92 statements in this chapter are for the embedded SQL-92 environment only. The elements and statements described in this chapter are:

(1 of 2)

BEGIN-END DECLARE SECTION	CLOSE Statement
DECLARE CURSOR Statement	DESCRIBE BIND VARIABLES Statement
DESCRIBE SELECT LIST Statement	EXEC SQL Delimiter
EXECUTE Statement	EXECUTE IMMEDIATE Statement
FETCH Statement	OPEN Statement
PREPARE Statement	Query Expressions
Search Conditions	SELECT Statement for a Single Row

(2 of 2)

Type Specifications for Host Language Types	Type Specifications for Static Array Types
WHENEVER Statement	

BEGIN-END DECLARE SECTION

Declares variables and types used by the precompiler. Any variables you refer to in an embedded SQL-92 statement must be declared in a DECLARE SECTION. This section starts with a BEGIN DECLARE SECTION statement and ends with an END DECLARE SECTION statement. Each variable must be declared as a host language data type.

SYNTAX

```
EXEC SQL BEGIN DECLARE SECTION
host_lang_type variable_name ;
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

host_lang_type variable_name ;

A conventional C Language variable declaration. This form of variable declaration conforms to the ANSI standard for the C Language.

SYNTAX

```
{ char | short | long | float | double }
```

EXAMPLE

This example is a code fragment from the `StatSel` function in the sample program 2StatSel.pc. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
EXEC SQL BEGIN DECLARE SECTION ;
      short InvTransNum_v ;
      short Qty_v ;
      short OrderNum_v ;
      EXEC SQL END DECLARE SECTION ;
```

NOTES

- The C Language type int is not supported by ESQLC. Type int maps to 16 or 32 bits depending on the machine architecture. This can create rounding errors at runtime as values are passed across different machine architectures.

- Variables you declare in a BEGIN-END DECLARE SECTION can be used in C Language statements as if they had been declared outside the DECLARE SECTION.
- The scope of variables follows host language scoping rules. The ESQLC variables are not visible outside the file in which they are declared.
- DECLARE sections are permissible only where host language declarations are permissible in the host language syntax. This restriction is due to how DECLARE SECTION blocks are translated into the main body of host language declarations.
- Avoid DECLARE sections in header files that are included by more than one source file. This can cause duplicate variables with the same name.
- The form of the variable created by ESQLC for each type is specified so that it can be manipulated from host language statements. Declaring variables allows you to use them in both host language and embedded SQL-92 statements.

AUTHORIZATION

None

SQL COMPLIANCE

Declarations that use host-language types are SQL-92 compliant.

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

Type Specifications for Static Array Types

CLOSE Statement

Closing a cursor changes the state of the cursor from open to closed.

SYNTAX

```
EXEC SQL CLOSE cursor_name ;
```

cursor_name

An identifier named earlier in a DECLARE CURSOR statement and an OPEN CURSOR statement.

EXAMPLE

This example is a code fragment from the seldone routine in the sample program 4DynSel.pc. The complete source for the sample program in Appendix A, "ESQL-92 Sample Programs."

```
EXEC SQL CLOSE dyncur ;
EXEC SQL COMMIT WORK ;
```

NOTES

- Only a cursor in the open state can be set to the closed state.
- When a transaction ends, any cursors in the open state are automatically set to the closed state.
- When a cursor is in the closed state, you cannot perform FETCH, DELETE, or UPDATE operations using that cursor.
- It is good practice to close cursors explicitly.

AUTHORIZATION

None. See AUTHORIZATION for the OPEN Statement.

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

DECLARE CURSOR Statement, OPEN Statement, FETCH Statement, positioned UPDATE, positioned DELETE

DECLARE CURSOR Statement

Associates a cursor with a static query or a prepared dynamic query statement. The query or the prepared statement can have references to host variables.

SYNTAX

```
DECLARE cursor_name CURSOR FOR
  { query_expr  [ ORDER BY clause ]  [  FOR UPDATE clause  ]
    |  prepared_statement_name
  } ;
```

cursor_name

A name you assign to the cursor. The name must meet the requirements for an identifier.

query_expr [ORDER BY *clause*] [FOR UPDATE *clause*]

A complete query expression.

prepared_statement_name

The name assigned to a prepared SQL-92 statement in an earlier PREPARE statement.

EXAMPLES

The first example is a code fragment from the `StatSel` function in the sample program 2StatSel.pc. This example illustrates the static processing of a SELECT statement The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL DECLARE stcur CURSOR FOR
     SELECT InvTransNum, Qty, OrderNum FROM PUB.InventoryTrans ;
EXEC SQL OPEN stcur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

This example is a code fragment from the `DynSel` function in sample program 4DynSel.pc. This example illustrates the dynamic processing of a SELECT statement. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

NOTES

- You must declare a cursor before any OPEN, FETCH, or CLOSE statement.
- The scope of the cursor declaration is the entire source file in which it is declared. The operations on the cursor such as OPEN, CLOSE, and FETCH can occur only within the same compilation unit as the cursor declaration.
- The use of a cursor allows the execution of the positioned forms of the UPDATE and DELETE statements.
- If the DECLARE statement corresponds to a static SQL statement with parameter references:
  - The DECLARE statement must be executed before each execution of an OPEN statement for the same cursor.
  - The DECLARE statement and the OPEN statement that follows must occur within the same transaction within the same task.
  - If the statement contains parameter references to automatic variables or function arguments, the DECLARE statement and the following OPEN statement for the same cursor must occur within the same C function.
- See the SELECT Statement for a Single Row reference entry for descriptions of the ORDER BY and FOR UPDATE clauses.

AUTHORIZATION

None. See AUTHORIZATION for the OPEN Statement.

SQL COMPLIANCE

SQL-92. Progress Extension: *`prepared_statement_name`*

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

PREPARE Statement, OPEN Statement, FETCH Statement, CLOSE Statement, SELECT Statement for a Single Row, Query Expressions

DESCRIBE Statements

Writes information about a prepared statement to the SQL Descriptor Area (SQLDA). You use a DESCRIBE statement in a series of steps that allows a program to accept SQL-92 statements at run time. Dynamically generated statements are not part of a program's source code; they are generated at run time.

There are two forms of the DESCRIBE statement:

- The DESCRIBE BIND VARIABLES Statement writes information about input variables in an expression to an SQLDA. These variables can be substitution variable names or parameter markers.
- The DESCRIBE SELECT LIST Statement writes information about select list items in a prepared SELECT statement to an SQLDA.

The SQLDA is a host language data structure used in dynamic SQL-92 processing. DESCRIBE statements write information about the number, data types, and sizes of input variables or select list items to SQLDA structures. Program logic then processes that information to allocate storage. OPEN, EXECUTE, and FETCH statements read the SQLDA structures for the addresses of the allocated storage.

DESCRIBE BIND VARIABLES Statement

Specifies that the DESCRIBE statement should write information about any input variables in the prepared statement to an input SQLDA structure. Input variables represent values supplied to INSERT and UPDATE statements at run time, and to predicates in DELETE, UPDATE, and SELECT statements at run time.

The DESCRIBE BIND VARIABLES statement writes the number of input variables to the `sqld_nvars` field of the SQLDA. If the `sqld_size` field of the SQLDA is not equal to or greater than this number, DESCRIBE writes the value as a negative number to `sqld_nvars`. Design your application to check `sqld_nvars` for a negative number to determine if a particular SQLDA is large enough to process the current input statement.

Input variables in dynamic SQL-92 statements are identified by parameter markers or as substitution names. See the PREPARE Statement reference entry for more information.

Design your program to issue a DESCRIBE BIND VARIABLES statement after a PREPARE statement, but before the corresponding EXECUTE or OPEN cursor statement for the SQL-92 statement.

This is the syntax for a DESCRIBE BIND VARIABLES statement:

SYNTAX

```
EXEC SQL
  DESCRIBE BIND VARIABLES FOR statement_name  INTO input_sqlda_name ;
```

statement_name

The name of an input SQL-92 statement to be processed using dynamic SQL-92 steps. Typically, this is the same *statement_name* used in the PREPARE statement.

input_sqlda_name

The name of the SQLDA structure to which DESCRIBE will write information about input variables.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

PREPARE Statement, DECLARE CURSOR Statement, OPEN Statement, FETCH Statement, CLOSE Statement, DESCRIBE SELECT LIST Statement

DESCRIBE SELECT LIST Statement

Specifies that the DESCRIBE statement should write information about select list items in a prepared SELECT statement to an output SQLDA structure. Select list items are column names and expressions in a SELECT statement. A FETCH statement writes the values returned by a SELECT statement to the addresses stored in an output SQLDA.

A DESCRIBE SELECT LIST statement writes the number of select list items to the `sqld_nvars` field of an output SQLDA. If the `sqld_size` field of the SQLDA is not equal to or greater than this number, DESCRIBE writes the value as a negative number to `sqld_nvars`. Design your application to check `sqld_nvars` for a negative number to determine if a particular output SQLDA is large enough to process the current SELECT statement.

Design your application to issue a DESCRIBE SELECT LIST statement after the DECLARE CURSOR, PREPARE, and OPEN statements for a dynamic SELECT statement, but before the first corresponding FETCH statement for the cursor.

This is the syntax for a DESCRIBE SELECT LIST statement:

SYNTAX

```
EXEC SQL
  DESCRIBE SELECT LIST FOR statement_name INTO output_sqlda_name ;
```

statement_name

The name of a SELECT statement to be processed using dynamic SQL-92 steps. Typically, this is the same *statement_name* as in the PREPARE statement.

output_sqlda_name

The name of the SQLDA structure to which DESCRIBE will write information about select list items.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

PREPARE Statement, DECLARE CURSOR Statement, OPEN Statement, FETCH Statement, CLOSE Statement, DESCRIBE BIND VARIABLES Statement

EXEC SQL Delimiter

In C Language programs, you must precede embedded SQL-92 statements with the EXEC SQL delimiter so that the precompiler can distinguish statements from the host language statements. Constructs within a BEGIN-END DECLARE SECTION do not require the EXEC SQL delimiter.

SYNTAX

```
EXEC SQL sql_statement ;
```

sql_statement

An SQL-92 statement to be processed by the ESQLC precompiler. You must terminate each SQL-92 statement with a semicolon to mark the end of the statement.

EXAMPLE

This example is a code fragment from the `DynSel` function in sample program 4DynSel.pc. This example illustrates the dynamic processing of a SELECT statement. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

NOTE

- In general, the ESQLC precompiler does not parse host language statements and therefore does not detect any syntax or semantic errors in host language statements. The exceptions to this rule are:
  - Recognition of host language blocks. The precompiler recognizes host language blocks in order to determine the scope of variables and types.
  - Constants defined with the #define preprocessor command. To evaluate these constants, the ESQLC precompiler invokes the C Language preprocessor before beginning embedded SQL-92 processing.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

None

EXECUTE Statement

Executes the statement specified in *statement_name*.

SYNTAX

```
EXEC SQL EXECUTE statement_name
  [ USING { [ SQL ] DESCRIPTOR structure_name
    | :host_variable [ [ INDICATOR ] :ind_variable ] , ... }
 ] ;
```

statement_name

The name of the prepared SQL-92 statement.

structure_name

The name of an SQL-92 descriptor area (SQLDA).

EXAMPLE

This example is a code fragment from the DynUpd function in sample program 3DynUpd.pc. This example illustrates the dynamic processing of an UPDATE statement The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
/*
** Process the non-SELECT input statement
**    PREPARE the statement
**    EXECUTE the prepared statement
**    COMMIT WORK
*/

      EXEC SQL PREPARE dynstmt FROM :sql_stmt_v ;
      EXEC SQL EXECUTE dynstmt ;
      EXEC SQL COMMIT WORK ;
```

NOTES

- A statement must be processed with a PREPARE Statement before it can be processed with an EXECUTE Statement.
- A prepared statement can be executed multiple times in the same transaction. Typically each call to the EXECUTE Statement supplies a different set of host variables.

- If there is no DESCRIPTOR in the USING clause, the EXECUTE Statement is restricted to the number of variables specified in the host variable list. The number and type of the variables must be known at compile time. The host variables must be declared in the DECLARE SECTION before they can be used in the USING clause of the EXECUTE Statement.
- If there is a DESCRIPTOR in the USING clause, the program can allocate space for the input host variables at run-time.

AUTHORIZATION

The user executing this statement must have authorization for the SQL statement being executed. See AUTHORIZATION for the relevant statement.

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

EXECUTE Statement, PREPARE Statement, SQLCA structure

EXECUTE IMMEDIATE Statement

Executes the statement specified in a *statement_string* or *host_variable*.

SYNTAX

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

statement_name

The name of the prepared SQL-92 statement.

structure_name

The name of an SQL-92 descriptor area (SQLDA).

NOTES

- The character string form of the statement is referred to as a *statement string*. An EXECUTE IMMEDIATE statement accepts either a statement string or a host variable as input.
- A statement string must not contain host variable references or parameter markers.
- A statement string must not begin with EXEC SQL and must not end with a semicolon.
- When an EXECUTE IMMEDIATE statement is executed, the SQL engine parses the statement and checks it for errors. Any error in the execution of the statement is reported in the SQLCA.
- If the same SQL-92 statement is to be executed multiple times, it is more efficient to use PREPARE and EXECUTE statements, rather than an EXECUTE IMMEDIATE statement.

AUTHORIZATION

The user executing this statement must have authorization for the SQL-92 statement being executed. See AUTHORIZATION for the relevant statement.

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

EXECUTE Statement, SQLCA structure

FETCH Statement

Moves the position of the cursor to the next row of the active set and fetches the column values of the current row into the specified host variables.

SYNTAX

```
EXEC SQL FETCH cursor_name
  { USING SQL DESCRIPTOR structure_name
    | INTO :host_var_ref [ [ INDICATOR ] :ind_var_ref ] , ...
  } ;
```

cursor_name

A name identified in an earlier DECLARE CURSOR Statement and an OPEN Statement.

USING SQL DESCRIPTOR *structure_name*

Directs the SQL engine to FETCH data into storage addressed by an SQLDA structure.

INTO :*host_var_ref* [[INDICATOR] :*ind_var_ref*]

Directs the SQL engine to FETCH data into the identified host variables, and to set values in the identified indicator variables.

EXAMPLE

This example is a code fragment from the `dynsel` function in the sample program 4DynSel.pc. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
/*
**  One way to limit the number of rows returned is to
**  set a new value for "j" here. As supplied in the SPORTS200 database,
**  the PUB.InventoryTrans table contains 75 rows.
*/
      j = 100;
      for (i = 0; i < j; i++)
           {
           EXEC SQL FETCH dyncur INTO
             :int_p1_v, :int_p2_v, :char_p_v ;
           if (i == 0)
               {
               printf ("\n\t 1st col \t\t 2nd col \t\t 3rd col");
               printf ("\n\t ------- \t\t ------- \t\t --------\n\n");
               }
           printf ("\t %d \t\t\t %d \t\t\t %s \n",
                   int_p1_v, int_p2_v, char_p_v) ;
           }
```

NOTES

- A FETCH operation requires that the cursor be open.
- The positioning of the cursor for each FETCH operation is as follows:
  - The first time you execute a FETCH Statement after opening the cursor, the cursor is positioned to the first row of the active set.
  - Subsequent FETCH operations advance the cursor position in the active set. The next row becomes the current row.
  - When the current row is deleted using a positioned DELETE statement, the cursor is positioned before the row after the deleted row in the active set.
- The cursor can only be moved forward in the active set by executing FETCH statements. To move the cursor to the beginning of the active set, you must CLOSE the cursor and OPEN it again.

- If the cursor is positioned on the last row of the active set or if the active set does not contain any rows, executing a FETCH will return the status code SQL_NOT_FOUND in the SQLCA.
- After a successful FETCH, the total row count fetched so far for this cursor is returned in `sqlca.sqlerrd[2]`. The count is set to zero after an OPEN cursor operation.
- You can FETCH multiple rows in one FETCH operation by using array variables in the INTO clause. The SQL_NOT_FOUND status code is returned in the SQLCA when the end of the active set is reached, even if the current FETCH Statement returns one or more rows.
- If you use array variables in a FETCH Statement the array sizes are set to the number of rows fetched after the FETCH Statement is executed.

AUTHORIZATION

None. See AUTHORIZATION for the OPEN Statement.

SQL COMPLIANCE

SQL-92. Progress Extension: USING DESCRIPTOR clause.

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

DECLARE CURSOR Statement, OPEN Statement, CLOSE Statement

OPEN Statement

Executes a prepared SQL-92 query associated with a cursor and creates a result set composed of the rows that satisfy the query. This set of rows is called the *active set*.

SYNTAX

```
EXEC SQL OPEN cursor_name
  [ USING { [ SQL ] DESCRIPTOR structure_name
    | :host_variable [ [ INDICATOR ] :ind_variable ] , ... }
  ] ;
```

cursor_name

An identifier named in an earlier DECLARE CURSOR Statement.

USING [SQL] DESCRIPTOR *structure_name*

Directs the SQL engine to create the result set in storage addressed by the identified SQLDA structure.

USING :*host_variable* [[INDICATOR] :*ind_variable*]

Directs the SQL engine to create the result set in storage addressed by host variables.

EXAMPLE

This example is a code fragment from the `StatSel` function in the sample program 2StatSel.pc. This example illustrates the static processing of a SELECT statement. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
/*
**    5.  Name WHENEVER routine to handle SQLERROR.
**
**    6.  DECLARE cursor for the SELECT statement.
**        NOTE: You must set input parameter values before OPEN CURSOR.
**        The static query in this program does not have input parameters.
**
**    7.  OPEN the cursor.
**        NOTE: For static statements, if a DECLARE CURSOR
**              statement contains references to automatic variables,
**              the OPEN CURSOR statement must be in the same C function.
**
**    8.  Name WHENEVER routine to handle NOT FOUND condition.
*/

      EXEC SQL WHENEVER SQLERROR GOTO selerr ;
      EXEC SQL DECLARE stcur CURSOR FOR
            SELECT InvTransNum, Qty,
            OrderNum FROM PUB.InventoryTrans ;
      EXEC SQL OPEN stcur ;
      EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

NOTES

- Executing an OPEN Statement sets the cursor to the open state.
- After the OPEN Statement is executed, the cursor is positioned just before the first row of the active set.
- For a single execution of an OPEN Statement, the active set does not change and the host variables are not re-examined.
- If you elect to retrieve a new active set and a host variable value has changed, you must CLOSE the cursor and OPEN it again.
- Execution of a COMMIT or ROLLBACK statement implicitly closes the cursors that have been opened in the current transaction.
- It is good practice to CLOSE cursors explicitly.

- When a cursor is in the open state, executing an OPEN Statement on that cursor results in an error.
- If a DECLARE CURSOR Statement is associated with a static SQL-92 statement containing parameter markers, the following requirements apply:
  - You must execute the DECLARE CURSOR Statement before executing the OPEN Statement for that cursor.
  - The DECLARE CURSOR Statement and the OPEN Statement for the same cursor must occur in the same transaction.
  - If the statement contains parameter markers for stack variables, the DECLARE CURSOR Statementand the following OPEN Statement for the same cursor must occur in the same C Language function.

AUTHORIZATION

The user executing this statement must have at least one of the following privileges:

- DBA privilege
- SELECT privilege on all the tables and views referenced in the SELECT statement associated with the cursor

SQL COMPLIANCE

SQL-92. Progress Extension: USING DESCRIPTOR clause

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

DECLARE CURSOR Statement, CLOSE Statement, FETCH Statement, positioned UPDATE, positioned DELETE

PREPARE Statement

Parses and assigns a name to an ad hoc or dynamically generated SQL-92 statement for execution. You use a PREPARE statement in a series of steps that allows a program to accept or generate SQL-92 statements at run time.

SYNTAX

```
EXEC SQL PREPARE statement_name FROM statement_string ;
```

statement_name

A name for the dynamically generated statement. The DESCRIBE Statements, EXECUTE Statement, and DECLARE CURSOR Statement refer to this *statement_name*. A *statement_name* must be unique in a program.

statement_string

Specifies the SQL-92 statement to be prepared for dynamic execution. You can either use the name of a C Language string variable containing the SQL-92 statement or specify the SQL-92 statement as a quoted literal.

SYNTAX

```
{ :host_variable | quoted_literal }
```

In either format, the statement string must be a character string that is a dynamically executable statement. If there is an SQL-92 syntax error, the PREPARE statement returns an error in the SQLCA.

EXAMPLES

The first example is a code fragment from the dynupd function in sample program 3DynUpd.pc, which illustrates dynamic processing of an UPDATE statement. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
/*
** Process a dynamic non-SELECT input statement
**    PREPARE the statement
**    EXECUTE the prepared statement
**    COMMIT WORK
*/

EXEC SQL PREPARE dynstmt FROM :sql_stmt_v ;
EXEC SQL EXECUTE dynstmt ;
EXEC SQL COMMIT WORK ;
```

This example is a code fragment from the dynsel function in sample program 4DynSel.pc, which illustrates dynamic processing of a SELECT statement. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
/*
**    PREPARE a the dynamic SELECT statement.
**    DECLARE cursor for the prepared SELECT statement.
**    NOTE: You must set input parameter values before OPEN CURSOR.
**    If your query has input parameters, you must define them in
**    the DECLARE SECTION.
**    OPEN the declared cursor.
**    NOTE: For static statements, if a DECLARE CURSOR
**    statement contains references to automatic variables,
**    the OPEN CURSOR statement must be in the same C function.
**
**    Name WHENEVER routine for NOT FOUND condition.
**    FETCH a row and print results until no more rows.
*/

EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

NOTES

- A statement string can have one or more references to input variables. These variables represent values supplied at run time to:
  - INSERT and UPDATE statements
  - Predicates in DELETE, UPDATE, and SELECT statements
- A program supplies an input variable to a PREPARE statement either as a *substitution name* or as a parameter marker:
  - A substitution name is a name preceded by a colon (:) in a statement string. This name does not refer to a C Language variable, but acts only as a placeholder for input variables.
  - A parameter marker is a question mark (?) in the statement string, serving as a placeholder for input variables.
- The USING clauses of EXECUTE Statement, EXECUTE IMMEDIATE Statement, and OPEN Statement identify host language storage. The values in this storage expand a statement string, replacing a substitution name or a parameter marker. You can design your program to execute the same prepared statement many times in a transaction, supplying different values for input variables for each execution. If you COMMIT or ROLLBACK the transaction, you must PREPARE the statement string again.

AUTHORIZATION

The user executing this statement must have at least one of the following privileges:

- DBA privilege.
- Authorization for SQL-92 statement being prepared. See the AUTHORIZATION section for the relevant statement.

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

EXECUTE Statement, EXECUTE IMMEDIATE Statement, OPEN Statement, CLOSE Statement, FETCH Statement, and SQLCA structure

Query Expressions

Identify a list of expressions whose results form columns in a result table. This list of expressions is called a *select list*.

NOTE: This information on query expressions is specific to ESQL-92. For additional information on query expressions, see the *Progress SQL-92 Guide and Reference*.

SYNTAX

```
SELECT expression [ ' column_title ' ]
  [ , expression [ ' column_title ' ] ] , ... ;
```

expression

A column name from a table in the FROM clause or any expression that returns a value. An expression can also be any supported mathematical expression, scalar function, or aggregate function that returns a value.

column_title

Optionally, specifies a new heading for a column in the result table.

EXAMPLE

This example is a code fragment from the `statsel` function in sample program 2StatSel.pc. This example illustrates a query expression and the static processing of a SELECT statement. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL DECLARE stcur CURSOR FOR
     SELECT InvTransNum, Qty, OrderNum FROM PUB.InventoryTrans ;
EXEC SQL OPEN stcur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

NOTES

- Typically, the *expression* is a column name from a table in the FROM clause. An expression can also be any supported mathematical expression, scalar function, or aggregate function that returns a value.
- When you specify the optional *column_title* you must enclose it in single quotes (').
- When a *column_title* is specified in a query expression, a DESCRIBE SELECT LIST Statement for a cursor associated with that SELECT Statement for a Single Row returns the *column_title* for the column name.

RELATED STATEMENTS

DESCRIBE Statements, SELECT Statement for a Single Row

Search Conditions

NOTE: This information on search conditions is specific to embedded SQL-92. For additional information on search conditions, see the *Progress SQL-92 Guide and Reference*.

LIKE Predicate

Searches a character string column for a specific pattern. The search pattern is specified in a string constant or in a host variable. In the search pattern string, the underscore (_) and the percent sign (%) have special semantics.

SYNTAX

```
column_name [ NOT ] LIKE { :host_variable | string_constant }
  [ ESCAPE escape_character ]
```

column_name

Identifies the column to be searched for a specific pattern.

:host_variable

Identifies a C Language variable containing the search pattern.

string_constant

Identifies a string constant containing the search pattern.

NOTES

- A *column_name* specified in the LIKE predicate must be of data type CHARACTER or data type VARCHAR.
- A *host_variable* specified in the LIKE predicate must be a character string variable.
- Use the ESCAPE clause to escape the special semantics given to the underscore (_) and percent (%) characters.
- The escape character must precede each special character in order to disable its semantics.

IN Predicate

Compares a value against a set of values.

SYNTAX

```
expression [ NOT ] IN
  { (query_expression)
    | ( { :host_variable | constant } [ ,... ] )
  }
```

expression

Identifies the value to be compared with a set of values.

query_expression

Indicates that the *expression* is to be compared with the single column value returned in the result set of a *query_expression*. Note that a *query_expression* in this context must be enclosed in parentheses.

({ *host_variable* | *constant* } [, ...])

Is a comma-separated list of one or more variables and constants against which the expression is to be compared. Note that the entire list must be enclosed in parentheses.

NOTE

- If an IN predicate specifies a *query_expression*, then the result table it returns should contain a single value. If the result table contains more than one value, an error results.

SELECT Statement for a Single Row

Returns no more than one row in the result table. Use this statement in embedded SQL-92 programs to assign the values of a single row result table directly to host variables. The syntax for the single row form is similar to the general form of a SELECT statement, with these limitations:

- An INTO clause is required
- SELECT * is not valid
- Set operators are not valid

SYNTAX

```
SELECT [ ALL | DISTINCT ]
  { expression [ [ AS ] [ ’] column_title [’ ] ]
    [, expr [ [ AS ] [ ’] column_title [’ ] ] ] , ...
  }
  INTO :host_variable [, :host_variable ] ,... [ { NO REORDER } ]
  FROM table_ref [, table_ref ] ... [ { NO REORDER } ]
  [ WHERE search_condition ]
  [ GROUP BY clause ]
  [ HAVING search_condition ]
  [ ORDER BY clause ]
  [ FOR UPDATE clause ]
  ;
```

`SELECT [ ALL | DISTINCT ]`

If you specify the `DISTINCT` keyword, duplicate rows will be eliminated from the rows retrieved by the system. The default action is that all selected rows are returned.

`INTO`

In the single row form of the `SELECT` statement, the selected values are assigned to the host variables specified in the `INTO` clause. The `INTO` clause is required, and it must precede the `FROM` clause. If no rows are selected, the SQL_NOT_FOUND condition is returned in the SQLCA structure.

FROM

The FROM clause is required. If the same column name exists in more than one table specified in the FROM clause, you must qualify the column name with the table name or a table alias name.

WHERE

Specifies the selection criteria for rows. If you do not specify a WHERE clause, all the rows of the Cartesian product of all the tables in the FROM clause are returned.

GROUP BY

Specifies the groups of rows returned by the SELECT statement. For more information, see the "Query Expressions" section in Chapter 2, "SQL-92 Language Elements," of the *Progress SQL-92 Guide and Reference*.

HAVING

Allows you to set conditions on the groups returned by a SELECT statement. For more information, see the "SELECT" reference section in Chapter 3, "SQL-92 Language Statements", of the *Progress SQL-92 Guide and Reference*.

ORDER BY

Allows you to specify the order in which the SELECT statement returns rows.

FOR UPDATE

Specifies which rows are to be updated.

NOTES

- For complete information on Search Conditions and Query Expressions, see Chapter 2, "SQL-92 Language Elements," in the *Progress SQL-92 Guide and Reference*.
- For complete information on the SELECT statement, see Chapter 3, "SQL-92 Statements," in the *Progress SQL-92 Guide and Reference*.

AUTHORIZATION

The user executing this statement must have at least one of the following privileges:

- DBA privilege
- SELECT privilege on all the tables referenced in the SELECT statement

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

None

Type Specifications for Host Language Types

Declares a variable or type that corresponds to a type supported by the host language. This host language type declaration must be within a BEGIN-END DECLARE SECTION.

SYNTAX

```
{ variable_name [ , ... ] | TYPE new_type_name }
  IS OF TYPE host_language_type ;
```

variable_name

C Language variable name.

TYPE *new_type_name*

The new type that corresponds to a type supported by the C Language.

IS OF TYPE *host_language_type*

Declares a variable or type that corresponds to a type supported by the host language.

SYNTAX

```
{ char | short | long | float | double }
```

NOTES

- The C Language type int is not supported by ESQLC. Type int maps to 16 or 32 bits depending on the machine architecture. This can create rounding errors at runtime as values are passed across different machine architectures.
- You can declare a variable to be of a specific type without using a host language type. For example, this variable declaration defines a variable of type long integer:

```
variable_name IS OF TYPE unsigned long;
```

- For convenience, you can define a new type which inherits all the properties of a host language type or a previously-declared type. You can use the new type to declare variables or other types in ESQLC statements or in C Language statements.
- You can declare ESQLC variables of C Language types in this form:

```
host_language_type variable_name ;
```

EXAMPLE

For example, the following declaration declares `customer_no` to be a variable of C Language type `unsigned long`.

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

ENVIRONMENT

Embedded SQL-92 only

Type Specifications for Static Array Types

Declares a static array or array type consisting of elements of any data type. This array declaration must be within a BEGIN-END DECLARE SECTION.

SYNTAX

```
{ variable_name [ , ... ] | TYPE new_type_name }
  IS AN ARRAY OF type_name WITH SIZE constant_id ;
```

variable_name

C Language variable name.

TYPE *new_type_name*

Identifies the data type of the array being declared.

IS AN ARRAY OF *type_name*

Identifies the data type of the array being declared.

WITH SIZE *constant_id*

A constant defining the dimension or size of the array.

NOTES

- ESQLC does not allow the definition of open size arrays.
- All ESQLC arrays except for character arrays are mapped into a host language structure consisting of both the actual array and the current size of the array. The C Language structure is of this form:

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

- Host language statements can manipulate the array assuming that is is a structure with the same name as the array name, and having two components `tpe_array` and `tpe_size`. The `tpe_size` component contains the actual array and the `tpe_size` component contains the current size of the array.

- All ESQLC statements that manipulate the array update the current size associated with the array. The host language statements that manipulate the array are expected to update the current size.

- Character arrays are a special case for the ESQLC precompiler. They are not mapped to a record, but rather as a null terminated string. This simplifies their manipulation from C and their use with system functions. When the ESQLC precompiler manipulates a character array, the null terminator is assured.

- When you refer to an array element in an SQL-92 statement, you should specify the array name followed by the element index in square brackets.

EXAMPLE

This example illustrates how to refer to an array element in an SQL-92 statement:

```
EXEC SQL BEGIN DECLARE SECTION ;
#define     NAMESZ     20
#define     ARRAYSZ    10
cust_name_t IS AN ARRAY OF char WITH SIZE NAMESZ ;
cust_name   IS AN ARRAY of cust_name_t WITH SIZE ARRAYSZ ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
for ( i = 0 ; i < ARRAYSZ ; i++ )
     {
     EXEC SQL FETCH cust_cursor INTO :cust_name[i] ) ;
     if (sqlca.sqlcode) break ;
     printf ("Customer name is %s \n", cust_name.tpe_array[i]) ;
     }
```

- If you specify array variables in the INTO clause of a FETCH statement, the ESQLC precompiler retrieves multiple rows. When you use an array for a multiple-row fetch, the sizes of all array variables must be the same. If they are different, the ESQLC precompiler uses the minimum of the array sizes.

EXAMPLE

This example illustrates an array variable in the INTO clause of a FETCH statement:

```
EXEC SQL BEGIN DECLARE SECTION ;
cust_name_t IS AN ARRAY OF char WITH SIZE 20 ;
cust_name   IS AN ARRAY of cust_name_t WITH SIZE 10 ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
EXEC SQL FETCH cust_cursor INTO :cust_name ;
```

- If you use a FETCH statement to fetch multiple rows into array variables, the array sizes are set appropriately after the fetch.

EXAMPLE

This example illustrates how to define an array variable to be used by a FETCH statement

```
EXEC SQL BEGIN DECLARE SECTION ;
TYPE customer_array_type IS AN ARRAY OF customer_record WITH SIZE 50 ;
EXEC SQL END DECLARE SECTION ;
   .
   .
   .
```

ENVIRONMENT

Embedded SQL-92 only

WHENEVER Statement

Specifies actions for three SQL-92 run-time exceptions.

SYNTAX

```
WHENEVER
  { NOT FOUND  |  SQLERROR  |  SQLWARNING }
  {  STOP  |  CONTINUE  |  {  GOTO  |  GO TO }  host_lang_label  } ;
```

NOT FOUND

The NOT FOUND exception is set when sqlca.sqlcode is set to SQL_NOT_FOUND.

SQLERROR

The SQLERROR exception is set when sqlca.sqlcode is set to a negative value.

SQLWARNING

The SQLWARNING exception is set when sqlca.sqlwarn[0] is set to 'W' after a statement is executed.

STOP

Results in the ESQLC program stopping execution.

CONTINUE

Results in the ESQLC program continuing execution. The default exception is to CONTINUE.

{ GOTO | GO TO } *host_lang_label*

Results in the ESQLC program execution to branch to the statement corresponding to the *host_lang_label*.

EXAMPLES

The first example is a code fragment from the `main` function in sample program 1StatUpd.pc. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
/*
**  Name WHENEVER routine to handle SQLERROR condition.
*/
EXEC SQL WHENEVER SQLERROR GOTO mainerr ;
```

This example is a code fragment from the `dynupd` function in sample program 3DynUpd.pc, which illustrates dynamic processing of an UPDATE statement. The complete source for the sample program is listed in Appendix A, "ESQL-92 Sample Programs."

```
/*
**    Name WHENEVER routines to handle NOT FOUND and SQLERROR
*/

EXEC SQL WHENEVER SQLERROR GOTO nodyn ;
EXEC SQL WHENEVER NOT FOUND GOTO nodyn ;
```

NOTES

- You can place multiple WHENEVER statements for the same exception in a source file. Each WHENEVER statement overrides the previous WHENEVER statement specified for the same exception.
- Correct operation of a WHENEVER statement with a GOTO *host_language_label* or a GO TO *host_language_label* is subject to the scoping rules of the C Language. The *host_language_label* must be within the scope of all SQL-92 statements for which the action is active. The GO TO or GOTO action is active starting from the corresponding WHENEVER statement until another WHENEVER statement for the same exception, or until end of file.

AUTHORIZATION

None.

SQL COMPLIANCE

SQL-92. Progress Extensions: SQLWARNING exception condition and STOP action.

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

FETCH Statement

A

ESQL-92 Sample Programs

This appendix provides the complete code listings for sample programs that contain or process embedded SQL-92 statements. Table A–1 lists the sample program names and summarizes what element of Embedded SQL-92 programing each source program illustrates.

Table A–1: Sample Programs

Program Name	Processing Element Illustrated
1StatUpd.pc	Static SQL-92 UPDATE (non-SELECT) statement
2StatSel.pc	Static SQL-92 SELECT statement
3DynUpd.pc	Dynamic SQL-92 non-SELECT statement
4DynSel.pc	Dynamic SQL-92 SELECT statement

These sample programs are provided with the installation media.

Each sample program is designed to access a copy of the SPORTS2000 sample database. As supplied, the static programs with SQL-92 statements already embedded access the PUB.InventoryTrans table. Descriptions of the steps in each program are in the source code. See Appendix B, "Compile, Link, and Run-time Requirements," for information on requirements for building these sample programs or any Embedded SQL-92 program.

1StatUpd.pc

This section contains the complete code listing of an ESQL-92 program with an embedded static SQL-92 UPDATE statement.

1StatUpd.pc *(1 of 6)*

```
/*****************************************************************************
** File Name: 1StatUpd.pc                                                   **
**                                                                          **
** Purpose: Illustrate processing of a static UPDATE statement.            **
**                                                                          **
**          Other static non-SELECT statements require similar             **
**          processing.                                                     **
**                                                                          **
** Overview:                                                                **
**   - Accepts connect_string, and optionally a transaction number.         **
**   - Updates the Qty column in the PUB.InventoryTrans table,              **
**     incrementing by 1.                                                   **
**   - As supplied, you can use sample program 2StatSel.pc to verify        **
**     results after executing this program.                                **
**   - You can modify this program, changing the statement to modify        **
**     different column(s) in a different table.                             **
**                                                                          **
** SQL Steps, in logical order:                                             **
** ============================                                             **
**   main()                                                                 **
**     1. DECLARE variable needed for connecting to a database.             **
**     2. Name WHENEVER routine to handle SQLERROR condition.               **
**     3. CONNECT to database specified in connect_string.                  **
**                                                                          **
**    11. After UPDATE operation, DISCONNECT from the database.             **
**                                                                          **
**   StatUpd()                                                              **
**     4. DECLARE variable(s) needed in UPDATE query.                       **
**     5. Name WHENEVER routine to handle NOT FOUND and SQLERROR.           **
**     6. EXECUTE the UPDATE statement.                                     **
**     7. COMMIT if UPDATE successful.                                      **
**     8. Examine sqlca.sqlcode for exception conditions.                   **
**     9. Direct processing to CONTINUE after SQLERROR.                     **
**    10. If SQLERROR, ROLLBACK transaction.                                **
**                                                                          **
** Requirements for building executable program:                            **
** =============================================                            **
**   - Precompile, compile, and link,                                       **
**     creating *.exe                                                       **
```

1StatUpd.pc *(2 of 6)*

```
**                                                                     **
**   - EXAMPLE for building in 1 step:                                 **
**     > esqlc 1StatUpd.pc -o 1StatUpd                                 **
**                                                                     **
** Requirements for executing:                                        **
** ===========================                                        **
**   - Copy of SPORTS2000 database, OR                                 **
**     Database containing InventoryTrans table, owner PUB             **
**     InventoryTrans table includes columns:                          **
**        InvTransNum (datatype short)                                 **
**        Qty (datatype short)                                         **
**   - Database running.                                               **
**   - Invoker must have authority to UPDATE PUB.InventoryTrans.       **
**   - To invoke from command line:                                    **
**     > 1statupd {connect_string} [Transaction_number]                **
**   - INPUTS (positional)                                             **
**       connect_string, in URL form (required).                       **
**       Transaction number, to match InvTransNum column (optional).   **
**       As supplied, the InvTransNum column in the                    **
**       Pub.InventoryTrans table has values ranging from 4 to 106.    **
**   - OUTPUTS                                                         **
**       Message to stdout, reporting success, NOT FOUND,              **
**       or exception condition.                                       **
**   - EXAMPLES for invoking from command line:                        **
**     (1) 1StatUpd progress:T:localhost:1024:y2ksports 55             **
**     (1) 1StatUpd progress:T:localhost:800:newsports 71              **
** Side effects:                                                      **
** =============                                                      **
**   - Successful execution updates the Qty column in the             **
**     PUB.InventoryTrans table.                                       **
**   - Locks held while transaction is active.                         **
**                                                                     **
*************************************************************************
**                                                                     **
** copyright (c) Progress Software Corporation, Bedford MA, 1999      **
**                     All rights reserved.                            **
**                                                                     **
** Revision History:                                                  **
**                                                                     **
** Date      By              Revision                                 **
** ====      ==              ========                                 **
** 10/1999   DB Doc          ESQL Sample Program for V9.1A            **
**                                                                     **
*************************************************************************/
```

1StatUpd.pc *(3 of 6)*

```
static void StatUpd(int TransNum) ;
static void usage_err() ;

struct  sqlca sqlca;
dh_i32_t    SQLCODE;

main (int argc, char **argv)
{
/*
**    1. DECLARE variable needed for connecting to a database
*/
      EXEC SQL BEGIN DECLARE SECTION ;
      char    connect_string_v[120] ;
      EXEC SQL END DECLARE SECTION ;

      int     TransNum ;

      if ((argc != 2) && (argc != 3))
          {
          usage_err () ;
          }

/*
** Save input parameters:
**       - connect_string
**       - if supplied, save transaction_number,
**         else default to transaction number 81.
*/

      strcpy (connect_string_v, argv[1]) ;
      if (argc == 3)
          {
          TransNum = atoi(argv[2]) ;
          }
      else
          {
          TransNum = 81 ;
          }
```

1StatUpd.pc *(4 of 6)*

```
/*
**  2. Name WHENEVER routine to handle SQLERROR condition.
**  3. CONNECT to the database named in connect_string.
**
** 11. After UPDATE operation, DISCONNECT from the database.
*/

      EXEC SQL WHENEVER SQLERROR GOTO mainerr ;
      EXEC SQL CONNECT TO :connect_string_v AS 'conn1' ;
      StatUpd(TransNum) ;
      EXEC SQL DISCONNECT 'conn1' ;/*
** If control passes here, UPDATE operation completed with no SQL errors.
*/
      exit (0) ;

/*
** Error Handling; control passes here
** WHENEVER there is an SQLERROR in main()
*/
mainerr:
      if (sqlca.sqlcode)
          {
          printf ("SQL Error (%ld) %s", sqlca.sqlcode, sqlca.sqlerrm) ;
          exit (1) ;
          }

} /* end main() */

/*****************************************************************************
**
** StatUpd(TransNum): process a static UPDATE statement.
**
*****************************************************************************/

static void StatUpd (int TransNum)
{
      char  errmesg[80];
/*
**    4. DECLARE variable needed in UPDATE statement.
*/
      EXEC SQL BEGIN DECLARE SECTION ;
      short InvTransNum_v ;
      EXEC SQL END DECLARE SECTION ;
```

1StatUpd.pc *(5 of 6)*

```
/*
**    Set inventory transaction number.
**
**    5. Define WHENEVER routine to handle NOT FOUND and SQLERROR
*/
      InvTransNum_v = TransNum ;

       EXEC SQL WHENEVER SQLERROR GOTO noupdate ;
      EXEC SQL WHENEVER NOT FOUND GOTO noupdate ;
/*
** Update Qty in Inventory Transaction (PUB.InventoryTrans) table
**     6. Execute the UPDATE statement
**     7. COMMIT if UPDATE successful
*/

      EXEC SQL
          UPDATE PUB.InventoryTrans
          SET Qty = Qty + 1
          WHERE InvTransNum = :InvTransNum_v ;
      EXEC SQL COMMIT WORK ;

      printf ("Static UPDATE statement executed successfully.") ;
      printf ("For InvTransNum %d, Qty value incremented by 1.",
               InvTransNum_v) ;
      return ;

      noupdate:
/*
**     8. Examine sqlca.sqlcode for exception conditions.
**     9. Direct processing to CONTINUE after SQLERROR.
**    10. If SQLERROR, ROLLBACK transaction.
*/
      if (sqlca.sqlcode == SQL_NOT_FOUND)
          {
          printf ("InvTransNum %d not found in InventoryTrans table ",
          InvTransNum_v);
          }

      if (sqlca.sqlcode < 0)
          {
          strncpy (errmesg, sqlca.sqlerrm, sqlca.sqlerrml);
          errmesg[sqlca.sqlerrml] = '' ;
          printf ("SQL Error : %s", errmesg);
          }
```

1StatUpd.pc *(6 of 6)*

```
      EXEC SQL WHENEVER SQLERROR CONTINUE;
      EXEC SQL ROLLBACK WORK ;
      exit (1);
 } /* end 1StatUpd() */

static void usage_err ()
{
    printf ("Usage: 1StatUpd <connect_string> [InvTransNum] ") ;
    printf ("   connect_string (required) ") ;
    printf ("   InvTransNum (optional) ") ;
    printf ("   Additional parameters are invalid ") ;
    exit (1) ;

} /* end usage_err() */
  /* end 1StatUpd.pc */
```

2StatSel.pc

This section provides the complete code listing of an ESQL-92 program that contains an embedded static SQL-92 SELECT statement.

2StatSel.pc *(1 of 6)*

```
/*****************************************************************************
** File Name: 2StatSel.pc                                                   **
**                                                                          **
** Purpose: Illustrate processing of a static SELECT statement.             **
**                                                                          **
** Overview:                                                                **
**   - Accepts a connect_string.                                            **
**   - Issues a SELECT statement, selecting the InvTransNum, Qty,           **
**     and OrderNum columns for all rows in the PUB.InventoryTrans          **
**     table.                                                               **
**   - As written, you can use this program to verify results from          **
**     the 1StatUpd.pc sample program.                                      **
**   - You can modify this program to issue a query against a               **
**     different database and table.                                        **
**                                                                          **
** SQL Steps, in logical order:                                             **
** ============================                                             **
**   main()                                                                 **
**     1. DECLARE variable(s) needed for connecting to a database.          **
**     2. Name WHENEVER routine to handle SQLERROR condition.               **
**     3. CONNECT to database specified in connect_string.                  **
**                                                                          **
**    14. After SELECT operation, DISCONNECT from database.                 **
**                                                                          **
**   StatSel()                                                              **
**     4. DECLARE output variables needed in SELECT statement.              **
**     5. Name WHENEVER routine to handle SQLERROR.                         **
**     6. DECLARE cursor for the SELECT statement.                          **
**     7. OPEN the declared cursor.                                         **
**     8. Name WHENEVER routine for NOT FOUND condition.                    **
**     9. FETCH a row, report results until no more rows.                   **
**    10. CLOSE the cursor.                                                 **
**    11. If success, COMMIT WORK to release locks.                         **
**    12. Direct processing to CONTINUE after SQLERROR.                     **
**    13. If SQLERROR, ROLLBACK transaction to release locks.               **
**                                                                          **
```

2StatSel.pc *(2 of 6)*

```
** Requirements for building executable program:                          **
** ============================================                           **
**   - Precompile, compile, and link,                                     **
**     creating *.exe                                                     **
**                                                                        **
**   - EXAMPLE for building in 1 step:                                    **
**     > esqlc 2StatSel.pc -o 2StatSel.exe                                **
**                                                                        **
** Requirements for executing:                                            **
** ===========================                                            **
**   - Copy of SPORTS2000 database, or                                    **
**     Database containing InventoryTrans table, owner PUB                **
**     Table must contain columns:                                        **
**        InvTransNum (short)                                             **
**        Qty (short)                                                     **
**        OrderNum (short)                                                **
**   - Database running                                                   **
**   - Invoker must have authority to access PUB.InventoryTrans           **
**   - To invoke from command line:                                       **
**     > 2StatSel {connect_string}                                        **
**   - INPUT                                                              **
**       connect_string, in URL form (required)                           **
**   - OUTPUT                                                             **
**       Message to stdout, reporting success, NOT FOUND,                 **
**       or exception condition.                                          **
**       One display row for each table row retrieved in the query.       **
**                                                                        **
** Side effects:                                                          **
** =============                                                          **
**   - Returns 3 columns from all rows in the PUB.InventoryTrans          **
**     table.                                                             **
**   - Holds lock on the table while transaction is active.               **
**                                                                        **
************************************************************************
**                                                                        **
** copyright (c) Progress Software Corporation, Bedford MA, 1999,2000 **
**                     All rights reserved.                               **
**                                                                        **
** Revision History:                                                      **
**                                                                        **
** Date      By              Revision                                     **
** ====      ==              ========                                     **
** 10/1999   DB Doc          ESQL Sample Program for V9.1A                **
**                                                                        **
************************************************************************/
```

2StatSel.pc *(3 of 6)*

```
static void StatSel() ;
static void usage_err() ;

struct  sqlca sqlca;
dh_i32_t    SQLCODE;

main (int argc, char **argv)
{
/*
**    1. DECLARE variable needed for connecting to a database
*/
      EXEC SQL BEGIN DECLARE SECTION ;
      char    connect_string_v[120] ;
      EXEC SQL END DECLARE SECTION ;

      if (argc != 2)
          {
          usage_err () ;
          }
/*
** Save connect_string input parameter.
*/
      strcpy (connect_string_v, argv[1]) ;

/*
**  2. Name WHENEVER routine to handle SQLERROR condition.
**  3. CONNECT to the database named in connect_string.
**
** 14. After SELECT operation, DISCONNECT from the database.
*/

      EXEC SQL WHENEVER SQLERROR GOTO mainerr ;
      EXEC SQL CONNECT TO :connect_string_v AS 'conn1' ;
      StatSel() ;
      EXEC SQL DISCONNECT 'conn1' ;

/*
** If control passes here, SELECT operation completed with no SQL errors.
*/
      exit (0) ;
```

2StatSel.pc *(4 of 6)*

```
/*
** Error Handling; control passes here
** WHENEVER there is an SQLERROR in main()
*/
mainerr:
      if (sqlca.sqlcode)
          {
          printf ("SQL Error (%ld) %s", sqlca.sqlcode, sqlca.sqlerrm) ;
          exit (1) ;
          }

} /* end main() */
/*****************************************************************************
**
** StatSel(): process a static SELECT statement.
**
*****************************************************************************/

static void StatSel ()
{
/*
**    4. DECLARE variables needed in SELECT statement.
**       NOTE 1: By program convention, host variables end with "_v".
**       NOTE 2: Variables with datatype "int" are invalid
**               in a DECLARE SECTION.
*/
      char  errmesg[80] ;
      int   i,j ;

      EXEC SQL BEGIN DECLARE SECTION ;
      short InvTransNum_v ;
      short Qty_v ;
      short OrderNum_v ;
      EXEC SQL END DECLARE SECTION ;
```

2StatSel.pc *(5 of 6)*

```
/*
**    5.  Name WHENEVER routine to handle SQLERROR.
**
**    6.  DECLARE cursor for the SELECT statement.
**        NOTE: You must set input parameter values before OPEN CURSOR.
**        The static query in this program does not have input parameters.
**
**    7.  OPEN the cursor.
**        NOTE: For static statements, if a DECLARE CURSOR
**              statement contains references to automatic variables,
**              the OPEN CURSOR statement must be in the same C function.
**
**    8.  Name WHENEVER routine to handle NOT FOUND condition.
**    9.  Retrieve a row with a FETCH operation until no more rows.
*/

      EXEC SQL WHENEVER SQLERROR GOTO selerr ;
      EXEC SQL DECLARE stcur CURSOR FOR
            SELECT InvTransNum, Qty,
            OrderNum FROM PUB.InventoryTrans ;
      EXEC SQL OPEN stcur ;
      EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
/*
**  One way to limit the number of rows returned is to
**  set a new value for "j" here. As supplied in the SPORTS200 database
**  the PUB.InventoryTrans table contains 75 rows.
*/
      j = 100;
      for (i = 0; i < j; i++)
           {
           EXEC SQL FETCH stcur INTO
             :InvTransNum_v, :Qty_v, :OrderNum_v ;

           if (i == 0)
               {
               printf ("      InvTransNum   Qty     OrderNum");
               printf ("      -----------   ---     --------");
               }
           printf (" %d     %d       %d ",
                   InvTransNum_v, Qty_v, OrderNum_v) ;
           }
```

2StatSel.pc *(6 of 6)*

```
seldone:
/*
**   NOTE: The SQLCA is available for examination only while the
**         cursor is open.
**
**   10.  CLOSE cursor.
**   11.  COMMIT WORK to release locks.
**        A SELECT statement does not modify the database.
*/
       printf ("Static SELECT statement executed successfully.") ;
       printf ("Number of rows returned = %ld ", sqlca.sqlerrd[2]) ;

       EXEC SQL CLOSE stcur ;
       EXEC SQL COMMIT WORK ;

       return ;

selerr:
/*
**   12.  Direct processing to CONTINUE after error.
**   13.  If SQLERROR, ROLLBACK transaction to release locks.
*/
      if (sqlca.sqlcode < 0)
      {
           strncpy (errmesg, sqlca.sqlerrm, sqlca.sqlerrml);
           errmesg[sqlca.sqlerrml] = '' ;
           printf ("SQL Error : %s", errmesg);
      }

      EXEC SQL WHENEVER SQLERROR CONTINUE;
      EXEC SQL ROLLBACK WORK ;
      exit (1);

} /* end 2StatSel() */

static void usage_err ()
{
    printf ("Usage: 2StatSel <connect_string> ") ;
    printf ("   connect_string (required) ") ;
    printf ("   Additional parameters are invalid ") ;
    exit (1) ;

} /* end usage_err() */
  /* end 2StatSel.pc */
```

3DynUpd.pc

This section provides the complete code listing of an ESQL-92 program that processes a dynamic SQL-92 non-SELECT statement.

3DynUpd.pc *(1 of 6)*

```
/************************************************************************
** File Name: 3DynUpd.pc                                               **
**                                                                     **
** Purpose: Illustrate processing of a dynamic non-SELECT statement.   **
**                                                                     **
**          All dynamic non-SELECT statements require similar          **
**          processing. To add complexity, add accepting and           **
**          processing of INPUT variables.                             **
**                                                                     **
** Overview:                                                           **
**   - Requires 2 input parameters:                                    **
**    (1) connect_string                                               **
**    (2) non-SELECT SQL-92 statement, requiring no INPUT variables;   **
**         constants permitted.                                        **
**   - As supplied, you can use sample program 4DynSel.pc to verify    **
**     results after executing this program.                           **
**                                                                     **
** SQL Steps, in logical order:                                        **
** ============================                                        **
**   main()                                                            **
**     1. DECLARE variable needed for connecting to a database.        **
**     2. Name WHENEVER routine to handle SQLERROR condition.          **
**     3. CONNECT to database specified in connect string.             **
**                                                                     **
**    12. After non-SELECT operation, DISCONNECT from database.        **
**                                                                     **
**   DynUpd()                                                          **
**     4. DECLARE variable for dynamic non-SELECT statement string.    **
**     5. Name WHENEVER routine to handle NOT FOUND and SQLERROR.      **
**     6. PREPARE the statement.                                       **
**     7. EXECUTE the prepared statement.                              **
**     8. COMMIT transaction if operation successful.                  **
**     9. Examine sqlca.sqlcode for exception conditions.              **
**    10. Direct processing to CONTINUE after SQLERROR.                **
**    11. If SQLERROR, ROLLBACK transaction.                           **
**                                                                     **
```

3DynUpd.pc *(2 of 6)*

```
** Requirements for building executable program:                          **
** ============================================                           **
**   - Precompile, compile, and link,                                     **
**     creating *.exe                                                     **
**                                                                        **
**   - EXAMPLE for building in 1 step:                                    **
**     > esqlc 3DynUpd.pc -o DynUpd                                       **
**                                                                        **
** Requirements for executing:                                           **
** ===========================                                           **
**   - Your database running.                                             **
**     Can use copy of SPORTS2000 db, and issue UPDATE,                   **
**        INSERT, and DELETE statements against these columns in          **
**        PUB.InventoryTrans table:                                       **
**            InvTransNum (datatype integer)                              **
**            Qty (datatype integer)                                      **
**            OrderNum (datatype integer)                                 **
**   - Invoker must have authority to UPDATE target database.             **
**   - To invoke from command line:                                       **
**     > 3DynUpd {connect_string} {dynamic non-SELECT statement}          **
**   - INPUTS (positional)                                                **
**       connect_string, in URL form (required)                           **
**       dynamic non-SELECT statement, in double quotes (required)        **
**   - OUTPUTS                                                            **
**       Message to stdout, reporting success, NOT FOUND,                 **
**       or exception condition                                           **
**   - EXAMPLES of dynamic non-SELECT statements that fit                 **
**       the model of the sample programs:                                **
**     (1) "UPDATE PUB.InventoryTrans set Qty = 5                         **
**          WHERE InvTransNum = 81"                                       **
**     (2) "DELETE FROM PUB.InventoryTrans WHERE InvTransNum = 81"        **
**     (3) "INSERT INTO PUB.InventoryTrans                                **
**          (InvTransNum, Qty, OrderNum) values (81, 99, 185)"            **
** Side effects:                                                         **
** ==============                                                        **
**   - Successful execution modifies the target database.                 **
**   - Table is locked while transaction is active.                       **
**                                                                        **
```

3DynUpd.pc *(3 of 6)*

```
/*****************************************************************************
**                                                                          **
** copyright (c) Progress Software Corporation, Bedford MA, 1999            **
**                     All rights reserved.                                 **
**                                                                          **
** Revision History:                                                        **
**                                                                          **
** Date       By              Revision                                      **
** ====       ==              ========                                      **
** 10/1999    DB Doc          ESQL-92 Sample Program for V9.1A              **
**                                                                          **
*****************************************************************************/

static void DynUpd(char *input_stmt) ;
static void usage_err() ;

struct  sqlca sqlca;
dh_i32_t    SQLCODE;

main (int argc, char **argv)
{
/*
**    1. DECLARE variable needed for connecting to a database.
*/
      EXEC SQL BEGIN DECLARE SECTION ;
      char    connect_string_v[120] ;
      EXEC SQL END DECLARE SECTION ;

      char     input_stmt[255] ;

      if (argc != 3)
          {
          usage_err () ;
          }

/*
** Save input parameters:
**       - 1st param => connect_string_v
**       - 2nd param => input_stmt
*/

      strcpy (connect_string_v, argv[1]) ;
      strcpy (input_stmt, argv[2]) ;
```

3DynUpd.pc *(4 of 6)*

```
/*
**  2. Name WHENEVER routine to handle SQLERROR condition.
**  3. CONNECT to the database named in 1st param.
**
** 12. After non-SELECT operation, DISCONNECT from the database.
*/

      EXEC SQL WHENEVER SQLERROR GOTO mainerr ;
      EXEC SQL CONNECT TO :connect_string_v AS 'conn1' ;
      DynUpd(input_stmt) ;

      EXEC SQL DISCONNECT 'conn1' ;
/*
** If control passes here, dynamic operation completed with no SQL errors.
*/
      exit (0) ;
/*
** Error Handling; control passes here
** WHENEVER there is an SQLERROR in main()
*/
mainerr:
      if (sqlca.sqlcode)
          {
          printf ("SQL Error (%ld) %s", sqlca.sqlcode, sqlca.sqlerrm) ;
          exit (1) ;
          }

} /* end main() */

/*****************************************************************************
**
** DynUpd(char *input_stmt): process a dynamic non-SELECT statement.
**
*****************************************************************************/

static void DynUpd (char *input_stmt)
{
      char  errmesg[80];
/*
**   4. DECLARE variable for dynamic non-SELECT statement string.
*/
      EXEC SQL BEGIN DECLARE SECTION ;
      char sql_stmt_v[255] ;
      EXEC SQL END DECLARE SECTION ;
     strcpy(sql_stmt_v, input_stmt) ;
```

3DynUpd.pc *(5 of 6)*

```
/*
**
**    5. Name WHENEVER routine to handle NOT FOUND and SQLERROR.
*/

      EXEC SQL WHENEVER SQLERROR GOTO nodyn ;
      EXEC SQL WHENEVER NOT FOUND GOTO nodyn ;

/*
** Process the non-SELECT input statement
**    6. PREPARE the statement
**    7. EXECUTE the prepared statement
**    8. COMMIT WORK
*/

      EXEC SQL PREPARE dynstmt FROM :sql_stmt_v ;
      EXEC SQL EXECUTE dynstmt ;
      EXEC SQL COMMIT WORK ;

      printf ("Dynamic non-SELECT statement executed successfully.") ;

      return ;
/*
**
**    5. Name WHENEVER routine to handle NOT FOUND and SQLERROR.
*/

      EXEC SQL WHENEVER SQLERROR GOTO nodyn ;
      EXEC SQL WHENEVER NOT FOUND GOTO nodyn ;

/*
** Process the non-SELECT input statement
**    6. PREPARE the statement
**    7. EXECUTE the prepared statement
**    8. COMMIT WORK
*/

      EXEC SQL PREPARE dynstmt FROM :sql_stmt_v ;
      EXEC SQL EXECUTE dynstmt ;
      EXEC SQL COMMIT WORK ;

      printf ("Dynamic non-SELECT statement executed successfully.") ;

      return ;
```

3DynUpd.pc *(6 of 6)*

```
 nodyn:
/*
**   9. Examine sqlca.sqlcode for exception conditions.
**  10. Direct processing to CONTINUE after SQLERROR.
**  11. If SQLERROR, ROLLBACK transaction.
*/
printf ("DEBUG, entering nodyn ") ;

      if (sqlca.sqlcode == SQL_NOT_FOUND)
          {
          printf ("Requested row not found ") ;
          }

      if (sqlca.sqlcode < 0)
          {
          strncpy (errmesg, sqlca.sqlerrm, sqlca.sqlerrml);
          errmesg[sqlca.sqlerrml] = '' ;
          printf ("SQL Error : %s", errmesg);
          }

      EXEC SQL WHENEVER SQLERROR CONTINUE;
      EXEC SQL ROLLBACK WORK ;
      exit (1);

} /* end 3DynUpd() */

static void usage_err ()
{
    printf ("Usage: 3DynUpd <connect string> <SQL-92 non-SELECT stmt> ") ;
    printf ("   connect_string (required) ") ;
    printf ("   SQL-92 non_SELECT statement in quotes (required) ") ;
    printf ("   Additional parameters are invalid ") ;
    exit (1) ;

} /* end usage_err() */
  /* end 3DynUpd.pc  */
```

4DynSel.pc

This section provides the complete code listing of an ESQL-92 program that processes a dynamic SQL-92 SELECT statement.

4DynSel.pc *(1 of 7)*

```
/************************************************************************
** File Name: 4DynSel.pc                                               **
**                                                                     **
** Purpose: Illustrate processing of a dynamic SELECT statement.       **
**                                                                     **
** Overview:                                                           **
**   - Requires 2 input parameters:                                    **
**    (1) connect_string                                               **
**    (2) SELECT SQL-92 statement enclosed in double quotes,           **
**         naming 3 columns of datatypes: integer, integer, varchar.   **
**   - As written, you can use this program to verify results          **
**         from sample program 3DynUpd.pc.                             **
**                                                                     **
** SQL Steps, in logical order:                                        **
** ============================                                        **
**   main()                                                            **
**     1. DECLARE variable needed for connecting to a database.        **
**     2. Name WHENEVER routine to handle SQLERROR condition.          **
**     3. CONNECT to database specified in connect_string.             **
**                                                                     **
**    15. After SELECT operation, DISCONNECT from database.            **
**                                                                     **
**   DynSel()                                                          **
**     4. DECLARE variables needed in SELECT statement.                **
**     5. Name WHENEVER routine to handle SQLERROR.                    **
**     6. PREPARE the dynamic SELECT statement.                        **
**     7. DECLARE cursor for the prepared SELECT statement.            **
**     8. OPEN the declared cursor.                                    **
**     9. Name WHENEVER routine for NOT FOUND condition.               **
**    10. FETCH a row and report output until no more rows.            **
**    11. CLOSE cursor.                                                **
**    12. COMMIT WORK to release locks.                                **
**    13. Direct processing to CONTINUE after SQLERROR.                **
**    14. If SQLERROR, ROLLBACK WORK to release locks.                 **
**                                                                     **
```

4DynSel.pc *(2 of 7)*

```
**                                                                      **
** Requirements for building executable program:                        **
** ============================================                         **
**   - Precompile, compile, and link,                                   **
**     creating *.exe                                                   **
**                                                                      **
**   - EXAMPLE for building in 1 step:                                  **
**     > esqlc 4DynSel.pc -o DynSel                                     **
**                                                                      **
** Requirements for executing:                                          **
** ===========================                                          **
**   - Copy of SPORTS2000 database, OR                                  **
**     Database containing PUB.InventoryTrans table, OR                 **
**     Database with table containing integer and varchar columns.      **
**   - Database running                                                 **
**   - Invoker must have authority to access table in FROM clause.      **
**   - To invoke from command line:                                     **
**     > 4DynSel {connect_string} {"SELECT statement in quotes"}        **
**   - INPUTS (positional)                                              **
**       (1) connect_string, in URL form (required).                    **
**       (2) SELECT statement enclosed in double quotes (required).     **
**           Statement should name 2 columns of datatype integer,       **
**           and 1 column of datatype varchar.                          **
**                                                                      **
**           These are the integer and varchar columns in the           **
**           PUB.InventoryTrans table:                                  **
**                                                                      **
**           column name               datatype                       **
**           -----------               --------                       **
**           BinNum                    integer                        **
**           InvTransNum               integer                        **
**           Itemnum                   integer                        **
**           Ordernum                  integer                        **
**           PONum                     integer                        **
**           Qty                       integer                        **
**           WarehouseNum              integer                        **
**           InvType                   varchar                        **
**           Transtime                 varchar                        **
**                                                                      **
**   - OUTPUTS                                                          **
**       (1) Message to stdout, reporting success, NOT FOUND,           **
**       (2) 3 columns from every row in the table.                     **
**                                                                      **
```

4DynSel.pc *(3 of 7)*

```
**   - EXAMPLES of valid queries that fit the model of the           **
**      sample programs:                                             **
**      "Select BinNum, InvTransNum, InvType from Pub.InventoryTrans" **
**      "Select Itemnum, Ordernum, Transtime from Pub.InventoryTrans" **
**                                                                   **
** Side effects:                                                     **
** =============                                                     **
**   - Returns 3 columns from all rows in the specified table.       **
**   - Holds lock on the table while the transaction is active.      **
**                                                                   **
***********************************************************************
**                                                                   **
** copyright (c) Progress Software Corporation, Bedford MA, 1999     **
**                     All rights reserved.                          **
**                                                                   **
** Revision History:                                                 **
**                                                                   **
** Date      By              Revision                                **
** ====      ==              ========                                **
** 10/1999   DB Doc          ESQL Sample Program for V9.1A           **
**                                                                   **
***********************************************************************/

static void DynSel(char *sel_stmt) ;
static void usage_err() ;

struct  sqlca sqlca;
dh_i32_t    SQLCODE;

main (int argc, char **argv)
{
/*
**    1. DECLARE variable needed for connecting to a database
*/
      EXEC SQL BEGIN DECLARE SECTION ;
      char    connect_string_v[120] ;
      EXEC SQL END DECLARE SECTION ;

      char  sel_stmt[255] ;

      if (argc != 3)
          {
          usage_err () ;
          }
```

4DynSel.pc *(4 of 7)*

```
/*
** Save input parameters:
**    - 1st param => connect_string_v
**    - 2nd param => sel_stmt
*/
      strcpy (connect_string_v, argv[1]) ;
      strcpy (sel_stmt, argv[2]) ;

/*
**  2. Define WHENEVER statement to handle SQLERROR condition.
**  3. CONNECT to the database named in connect_string.
**
** 14. After SELECT operation, DISCONNECT from the database.
*/

      EXEC SQL WHENEVER SQLERROR GOTO mainerr ;
      EXEC SQL CONNECT TO :connect_string_v AS 'conn1' ;
      DynSel(sel_stmt) ;

      EXEC SQL DISCONNECT 'conn1' ;

/*
** If control passes here, SELECT operation completed with no SQL errors.
*/
      exit (0) ;

/*
** Error Handling; control passes here
** WHENEVER there is an SQLERROR in main()
*/
mainerr:
      if (sqlca.sqlcode)
          {
          printf ("SQL Error (%ld) %s", sqlca.sqlcode, sqlca.sqlerrm) ;
          exit (1) ;
          }

} /* end main() */
```

4DynSel.pc *(5 of 7)*

```
/*************************************************************************
**
** DynSel(): process a dynamic SELECT statement.
**
*************************************************************************/

static void DynSel (char *sel_stmt)
{
/*
**    4. DECLARE variables needed in SELECT statement.
**       NOTE 1: By program convention, host variables end with "_v".
**       NOTE 2: Variables with datatype "int" are invalid
**               in the DECLARE SECTION.
*/
      char  errmesg[80] ;
      int   i,j ;

      EXEC SQL BEGIN DECLARE SECTION ;
      char  sel_stmt_v[255] ;     /* INPUT SELECT stmt     */
      short int_p1_v ;            /* OUTPUT 1st int column */
      short int_p2_v ;            /* OUTPUT 2nd int column */
      char  char_p_v[100] ;       /* OUTPUT CHAR column    */
      EXEC SQL END DECLARE SECTION ;

/*
**    5.  Name WHENEVER routine to handle SQLERROR.
**    6.  PREPARE the dynamic SELECT statement.
**    7.  DECLARE cursor for the prepared SELECT statement.
**        NOTE: You must set input parameter values before OPEN CURSOR.
**        If your query has input parameters, you must define them in
**        the DECLARE SECTION.
**    8.  OPEN the declared cursor.
**        NOTE: For static statements, if a DECLARE CURSOR
**              statement contains references to automatic variables,
**              the OPEN CURSOR statement must be in the same C function.
**
**    9.  Name WHENEVER routine for NOT FOUND condition.
**   10.  FETCH a row and print results until no more rows.
*/

      strcpy(sel_stmt_v, sel_stmt) ;

      EXEC SQL WHENEVER SQLERROR GOTO selerr ;
      EXEC SQL PREPARE stmtid from :sel_stmt_v ;
      EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
      EXEC SQL OPEN dyncur ;
      EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

4DynSel.pc *(6 of 7)*

```
/*
**  One way to limit the number of rows returned is to
**  set a new value for "j" here. As supplied in the SPORTS200 database,
**  the PUB.InventoryTrans table contains 75 rows.
*/
      j = 100;
      for (i = 0; i < j; i++)
           {
           EXEC SQL FETCH dyncur INTO
             :int_p1_v, :int_p2_v, :char_p_v ;
           if (i == 0)
               {
               printf (" 1st col  2nd col  3rd col");
               printf (" -------  -------  --------");
               }
           printf (" %d  %d  %s ",
                   int_p1_v, int_p2_v, char_p_v) ;
           }

seldone:
/*
**   NOTE: The SQLCA is available for examination only while the
**         cursor is open.
**
**   11.  CLOSE the cursor.
**   12.  COMMIT the transaction to release locks.
*/

       printf ("Dynamic SELECT statement executed successfully.") ;
       printf ("Number of rows returned = %ld ", sqlca.sqlerrd[2]) ;

       EXEC SQL CLOSE dyncur ;
       EXEC SQL COMMIT WORK ;
       return ;
```

4DynSel.pc *(7 of 7)*

```
selerr:
/*
**   13.  Direct processing to CONTINUE after SQLERROR.
**   14.  If SQLERROR, ROLLBACK WORK to release locks.
*/
      if (sqlca.sqlcode < 0)
      {
           strncpy (errmesg, sqlca.sqlerrm, sqlca.sqlerrml);
           errmesg[sqlca.sqlerrml] = '' ;
           printf ("SQL Error : %s", errmesg);
      }
      EXEC SQL WHENEVER SQLERROR CONTINUE;
      EXEC SQL ROLLBACK WORK ;
      exit (1);
} /* end 4DynSel() */

static void usage_err ()
{
    printf ("Usage: 4DynSel <connect_string> <SELECT statement>") ;
    printf ("   connect_string (required) ") ;
   printf ("   SELECT stmt listing 3 columns in double quotes (required) ") ;
    printf ("   Additional parameters are invalid ") ;
    exit (1) ;

} /* end usage_err() */
  /* end 4DynSel.pc  */
```

B

Compile, Link, and Run-time Requirements

This appendix lists compile, link, and run-time requirements for using the Embedded SQL-92 precompiler for C. Some requirements apply to all platforms; requirements for specific platforms are listed in tabular form.

The list of supported platforms and related development environments continues to expand. For the most current information related to the Embedded SQL-92 precompiler for C, see Progress Software Technical Support Knowledge Base entry #18914.

B.1 Requirements for All Platforms

The ESQL-92 precompiler for C has specific requirements for compiling, linking, and run time. These requirements vary across platforms. The run-time environment requires dynamic load library routines that are available in a C++ development environment. C Language development environments are not sufficient.

For all platforms, set DB_NAME, DH_USER, and DH_PASSWD environment variables to site-specific values. All platforms require a temporary directory at compile time. Define this with environment variable TPE_TMPDIR.

NOTE: For MS Windows NT only, do not enclose the connect string in single quotes in the definition of the DB_NAME environment variable.

B.2 Compile, Link, and Run-time Requirements by Platform

Table B–1 lists compile requirements, link requirements, and run-time requirements by platform.

Table B–1: Compile, Link, and Run-time Requirements by Platform *(1 of 2)*

Operating System Platform	Compile Requirements	Link Requirements	Run-time Requirements
MS Windows NT	MS Visual Studio 6 Include %DLC%\bin in PATH variable	esql92.lib esql92.dll Set LIB variable = %DLC%\lib	esql92.dll Include %DLC%\lib in PATH variable
MKS Windows NT (Korn Shell)	 Include $DLC/bin in PATH variable	esql92.lib esql92.dll Set LIB variable = $DLC/lib	esql92.dll Include $DLC/lib in PATH variable

Table B–1: Compile, Link, and Run-time Requirements by Platform *(2 of 2)*

Operating System Platform	Compile Requirements	Link Requirements	Run-time Requirements
Solaris	Sun Workshop Development Environment Include $DLC/bin in PATH variable	libesql92.so Set LD_LIBRARY_PATH variable = $DLC/lib	Include $DLC/jre/lib/solaris/native_threads in LD_LIBRARY_PATH variable
HPUX 1132	aCC Compiler Include location of compiler in PATH variable	libesql92.sl Include $DLC/lib and $DLC/usr/lib in LPATH variable	Include $DLC/lib:$DLC/jre/lib/PA_RISC/native_threads in SHLIB_PATH variable
AIX	 Include $DLC/bin in PATH variable	libesql92.so Set LIBPATH variable = $DLC/lib Set TPEROOT variable = $DLC	Include $DLC/dlc/jre/lib/aix/native_threads in LIBPATH variable

Glossary

Active Set

The collection of rows that SQL identifies when it opens a cursor and executes the query associated with the cursor. Also called a result set.

Atomicity

The property of transactions that either all the operations in a transaction are completed (if the transaction is committed) or none are completed (if the transaction is rolled back).

Candidate Key

Another term for UNIQUE key.

Commit

Make permanent all changes made during a transaction.

Constraint

Part of an SQL table or column definition that restricts the values that can be stored in a table or column. When you INSERT, DELETE, or UPDATE values, the constraint checks the new values against the conditions specified in the constraint. If the value violates the constraint, it generates an error. Constraints enforce REFERENTIAL INTEGRITY by ensuring that a value stored in the foreign key of a table must either be NULL or be equal to some value in the matching UNIQUE or PRIMARY KEY of another table.

Current Row

The current row of the active set of an open cursor. After an OPEN statement, the cursor is positioned just before the first row of the active set. The first FETCH statement advances the cursor position and makes the first row the current row. Subsequent FETCH statements make subsequent rows the current row.

Cursor

The active set defined by the query in a DECLARE CURSOR statement. Host programs use cursors to retrieve multiple rows of data returned by queries.

Cycle

The process of creating database tables when the first table has a foreign key that references the second table, the second table has a foreign key that references the third table and so on, and the last table has a foreign key that references the first table.

Data Control Language (DCL) Statements

SQL-92 GRANT and REVOKE statements. DCL statements control access to data and the rights to issue DCL statements.

Data Definition Language (DDL) Statements

SQL-92 CREATE, ALTER, and DROP statements used to manage tables, views, indexes, and other database objects.

Data Manipulation Language (DML) Statements

SQL-92 SELECT, INSERT, UPDATE, and DELETE statements that access or modify data.

Database Storage Representation

The format used to store a value in the database. For some data types this format is different than host language representation.

Durability

The characteristic of transactions that requires that all changes made during a transaction be permanent after the transaction is committed.

Dynamic SQL

A set of special SQL-92 statements (PREPARE, DESCRIBE, EXECUTE, and EXECUTE IMMEDIATE) and data structures (SQLCA and SQLDA) that allow programs to accept or generate SQL-92 statements at run time. Such dynamically-generated statements are not necessarily part of the program source code, but can be generated at run time.

Embedded SQL

SQL-92 statements that are embedded within a host language program. The ESQL precompiler translates embedded SQL-92 statements to equivalent C Language calls to routines in the Progress SQL-92 application programming interface.

ESQLC

The Progress Embedded SQL-92 precompiler for C host programs, and the command-line syntax to invoke it.

Exception Handler

Host program code that tests for a variety of possible errors and specifies what action will be taken if they arise.

Exclusive Locks

Locks that SQL-92 acquires on rows that have been modified by a transaction. Exclusive locks prevent other transactions from either reading or modifying the rows until the transaction is committed or rolled back.

Foreign Key

A column or columns in a table whose values must be either null or equal to some value in a corresponding column (called the primary key) in another table. Use the REFERENCES clause in the SQL CREATE TABLE statement to create foreign keys.

Host Language

Any programming language in which SQL statements can be embedded for database access. The ESQL precompiler supports embedding SQL statements in C Language programs.

Host Language Representation

The format used by a host language to represented values. This format is different from the database storage representation used by Progress SQL-92 for some data types. In some cases, applications must explicitly convert between data types to insert data or manipulate data retrieved from the database.

Host Program

An application program in which SQL-92 statements are embedded.

Host Variable

Any host language variable that is used in embedded SQL-92 statements. Programs must declare host variables in the DECLARE SECTION. Host variables can be declared as a database, host-language, or user-defined types. Depending on how they are used in an SQL-92 statement, host variables are either INPUT, where values stored by the program are used as an expression in an SQL-92 statement, or OUTPUT, where SQL-92 stores values returned from queries for use by the program.

Index

A database structure that speeds access to particular rows in a table. Indexes specify one or more columns as an index key. Typically queries that use the index key retrieve data faster than those that do not take advantage of an index.

Indicator Variable

A variable that is used to represent good value, truncation, or NULL VALUE in an application program. An indicator variable must be associated with a host variable. If the OUTPUT value retrieved by a query is null, SQL stores a negative in the indicator variable. Programs set INPUT indicator variables to -1 to specify a NULL VALUE for insertion into the database. Programs declare indicator variables as data type LONG or INTEGER.

Input Host Variable

A host variable that is used as INPUT to an embedded SQL-92 statement. Embedded SQL-92 statements can refer to host variables anywhere they can refer to an expression. Host programs use input variables to provide values to INSERT and UPDATE data in the database, and as arguments to search conditions.

Input SQLDA

An SQLDA (SQL Descriptor Area) used by a dynamic SQL-92 program to determine the number and data type of input parameters of an SQL-92 statement, and to store values for those input parameters. DESCRIBE, BIND VARIABLES, EXECUTE, and OPEN statements can specify an input SQLDA that SQL-92 uses to store information about input parameters.

Integrity Constraint

Another term for a category of constraints.

Isolation Level

The degree by which a transaction is isolated from the database modification operations of other concurrently active transactions.

Null Value

The absence of a value. Host programs use indicator variables or SQL-92 constructs to specify NULL VALUES and to retrieve NULL VALUES from the database.

Output Host Variable

A host variable that SQL-92 uses to store results from a query.

Output SQLDA

An SQLDA (SQL Descriptor Area) used by a dynamic SQL-92 program to determine the number and data type of columns in the result set returned by a query, and to retrieve values of that result set. DESCRIBE, SELECT LIST, and FETCH statements can specify an output SQLDA that SQL-92 uses to store the column information and data.

Parameter Marker

Question marks (?) in the statement string of a PREPARE statement. Parameter markers act as placeholders for input variables in dynamic SQL statements.

Precompiler

A translator that translates embedded SQL-92 statements in a host program to the equivalent C Language calls to functions in the Progress SQL-92 application programming interface (API).

Primary Key

A subset of the fields in a table, characterized by the constraint that no two records in a table can have the same PRIMARY KEY value, and that no fields of the PRIMARY KEY can have a NULL VALUE. You specify PRIMARY KEYS in a CREATE TABLE statement.

Referential Constraint

Another term for a category of constraints.

Referential Integrity

The condition where the value stored in a database table's FOREIGN KEY must either be NULL or be equal to some value in another table's matching UNIQUE or PRIMARY KEY. Constraints specified as part of CREATE TABLE statements prevent updates that violate referential integrity.

Relation

Another term for table.

Result Set

Another term for active set.

Result Table

A temporary table of values returned by a query.

Rollback

Undo all changes made during a transaction.

Share Locks

Locks that SQL-92 acquires on rows that have been read by a transaction. Share locks allow other transactions to read the row but prevent them from modifying the row until this transaction is either committed or rolled back.

SQLCA (SQL Communication Area)

A host structure that SQL-92 uses to store information about the execution of an SQL-92 statement. The SQLCA contains information about the most recently executed SQL statement.

SQLCODE

One of the fields in the SQLCA that contains a LONG INTEGER value indicating the status of the execution of an SQL-92 statement.

SQLDA (SQL Descriptor Area)

A host structure used in dynamic SQL-92 programs. The host program uses the SQLDA to determine the number and data types of INPUT and OUTPUT parameters of a dynamically-generated SQL-92 statement. SQL-92 uses the SQLDA to determine where to retrieve INPUT parameter values or to store OUTPUT query results.

Statement String

In dynamic SQL-92, the SQL-92 statement used as an argument to the EXECUTE IMMEDIATE and PREPARE statements.

Static SQL Statement

An SQL-92 statement that is known to the application at compile time, as opposed to a SQL-92 statement that is generated at run time.

Substitution Name

In the statement string of a PREPARE statement, a name preceded by a colon. Substitution names do not refer to host-language variable names, but act only as placeholders for INPUT variables in dynamic SQL-92 statements.

Table

The representation of data in a relational database as a collection of columns and rows. Also called a relation.

Transaction

A group of database operations whose changes can be made permanent or undone only as a unit.

Transaction Isolation Level

Another term for isolation level.

Unique Key

A column or columns in a table whose value (or combination of values) must be unique. Use the UNIQUE clause of the SQL-92 CREATE TABLE statement to create UNIQUE KEYS. UNIQUE KEYS are also called CANDIDATE KEYS.

View

A virtual table that re-creates the result table specified by a SELECT statement. No data is stored in a view, but other queries can refer to it as if it were a table containing data corresponding to the result table it specifies.

Index

A

Active set
  definition Glossary–1
  fetching rows 7–7

Aggregate functions
  ignoring NULL values 8–10

Array fetch
  definition 7–11
  dynamically allocated storage 10–16
  explicitly declared array 7–11
  extension of SQL-92 standard 7–13
  using an SQLDA 10–16

Atomicity
  definition Glossary–1
  property of a transaction 11–2

Audience xiii

B

BINARY data type 13–10

BIT data type 13–10

Bold typeface
  as typographical convention xvi

C

C Language compiler 2–6

C Language compiler commands
  UNIX platforms 2–5
  Windows NT platforms 2–5

Candidate key
  definition Glossary–1

Candidate key constraint 5–16 to 5–18
  column level 5–17
  table level 5–17

CHARACTER data type 13–2

Check constraint 5–11 to 5–13
  column level 5–12
  table level 5–13

CLOSE cursor statement 7–9, 10–8

Column
  default value 8–2

COMMIT statement 3–13
  definition Glossary–1
  ending a transaction 11–4

Computations 12–4

Condition evaluation 12–6
using SQL for 12–6

CONNECT statement 4–2 to 4–5
connecting to remote database 4–5
connecting to the DEFAULT database 4–5
using a connect string 4–3
using a connection name 4–3

Connection management statements
CONNECT 4–2
DISCONNECT 4–6
SET CONNECTION 4–5

Constraint
definition Glossary–1

Constraints 5–9
need for 5–9
types of 5–10
candidate key constraint 5–16
check constraint 5–11
primary key constraint 5–14
referential constraint 5–18
UNIQUE keyword 5–9, 5–16

CONTINUE action with the WHENEVER statement 9–11

CREATE INDEX statement 5–5

CREATE TABLE statement 5–2

CURRENT OF cursor construct 7–9

Current Row
definition Glossary–1

Cursor
associating with a query 7–5
closed state 7–5
closing 7–9
CURRENT OF 7–9
declaring 7–6
definition 7–5, Glossary–1
open state 7–5
opening 7–6
positioned delete 7–10
positioned update 7–10

Cycle
definition Glossary–2

Cycles
creating tables in 5–23
handling in referential integrity 5–21
inserting rows into tables 5–23

D

Data buffers
allocating 10–18
freeing 10–19

Data Control Language (DCL) 1–4
Statements
definition Glossary–2

Data conversion 13–11, 13–16
explicit comparison
using SQL and syscalctable 13–19
using tpe_compare_data() 13–16
explicit conversion 13–11
using scalar functions 13–14
using SQLand syscalctable 13–15
using tpe_conv_data() 13–13, 13–14
implicit conversion 13–11

Data Definition Language (DDL) 1–4
Statements
definition Glossary–2

Data Manipulation Language (DML) 1–4
Statements
definition Glossary–2

Data types 13–1 to 13–19
  categories of
    approximate numeric 13–7
    bit string 13–10
    character 13–2
    date-time 13–8
    exact numeric 13–4
  database storage representation 13–2
  host language representation 13–2
  symbolic names for 13–12
  using
    BINARY 13–10
    BIT 13–10
    DATE 13–8
    DECIMAL 13–5
    FLOAT 13–7
    INTEGER 13–4
    NUMERIC 13–6
    REAL 13–7
    SMALLINT 13–5
    TIME 13–9
    TIMESTAMP 13–10
    TINYINT 13–5
    VARBINARY 13–11
    VARCHAR 13–3

Database
  connection 4–2

Database access
  executable statements 3–3

Database Storage Representation
  definition Glossary–2

DECIMAL data type 13–5

DECLARE CURSOR statement 7–5

DECLARE SECTION 3–2 to 3–14
  limitations of
    declaring as STRUCTURE type 3–11
    referring to names in typedef statements 3–11

DECLARE statements 1–3
  definition 1–4
  EXEC SQL BEGIN DECLARE SECTION 1–4
  EXEC SQL DECLARE CURSOR 1–4
  EXEC SQL END DECLARE SECTION 1–4
  EXEC SQL WHENEVER 1–4

DECODE function 12–8

DELETE statement 6–5 to 6–6
  deleting all rows 6–5
  deleting one or more rows 6–5

DESCRIBE statement
  BIND VARIABLES FOR clause 10–20 to 10–22
  SELECT LIST FOR 10–20

DISCONNECT statement 4–6, 4–6 to 4–8
  using a connection name 4–7

DROP INDEX statement 5–6, 5–7

DROP TABLE statement 5–4

DROP VIEW statement 5–8

Durability
  definition Glossary–2
  property of a transaction 11–2

Dynamic non-SELECT statements A–14

Dynamic SELECT statement
  general steps for processing 10–6

Dynamic SELECT statements A–20

Dynamic SQL

Dynamic SQL processing 10–1 to 10–25
non-SELECT statement
general steps 10–2
SELECT statement
general steps 10–2

E

Embedded SQL
definition Glossary–2

Error handling 9–2 to 9–14
using indicator variables 9–14
using the SQLCA 9–2 to 9–9
using the WHENEVER statement 9–10

ESQL
advantages of using 1–3
application components 1–3
DECLARE SECTION 3–1
DECLARE statements 1–3
definition 1–2
executable statements 1–3, 3–3
library calls 1–4
program structure 3–1

ESQL constructs
definition 1–2

ESQL precompiler 2–3
EXEC SQL prefix 1–2

ESQLC
command syntax 2–3
definition 1–2, Glossary–2
ESQLC commnand 2–1
overview 1–4
source files 2–2
usage summary 2–2

ESQLC command options
+G option 2–4
+K option 2–4
+L option
suppress line numbers 2–4
+P option 2–4, 12–2
#define statements 12–2
preprocess 12–2
+T option
generate C source 2–4
+V option
verbose 2–4
-c
compile only 2–5, 2–6
-o
executable file name 2–5, 2–6
plus (+) prefix 2–3

ESQLC examples
+G option
print statements 2–9
+K option
keep intermediate files 2–8
+V verbose display 2–7

Exception Handler
definition Glossary–3

EXCLUSIVE locks
in transactions 11–8

EXEC SQL
prefix 1–2

EXEC SQL BEGIN DECLARE SECTION statement 1–4

EXEC SQL DECLARE CURSOR statement 1–4

EXEC SQL END DECLARE SECTION statement 1–4

EXEC SQL WHENEVER statement 1–4

Executable program
steps in building 2–6

Executable statements 1–3, 3–3
Data Control Language (DCL) 1–4, 3–11
Data Definition Language (DDL) 1–4, 3–11, 5–1
Data Manipulation Language (DML) 1–4, 3–11, 6–1
transaction management statements 3–11

EXECUTE IMMEDIATE statement 10–5
limitations 10–5

EXECUTE statement 10–4

F

FETCH statement 7–5, 7–7
using a cursor 10–6

File names
ESQLC programs 2–2

FLOAT data type 13–7

Foreign Key
definition Glossary–3

G

+G option
ESQLC command 2–4

GO TO action with the WHENEVER statement 9–11

H

Host Language
definition 1–2, Glossary–3

Host Language Representation
definition Glossary–3

Host Program
definition Glossary–3

Host variable
as database data type 3–3
data types 3–3
definition 3–2, Glossary–3
summary of how and when to use 3–5

Host variables
INPUT 6–8
OUTPUT 7–3

I

Index
definition Glossary–3

Indicator variable 3–2
definition 3–5, Glossary–4
null value handling 3–6
types 3–5
using with INPUT host variable 3–7
using with OUTPUT host variable 3–8

Indicator variable buffers
allocating 10–18
freeing 10–19

Indicator variables
for error handling 9–14
inserting NULL values into a column 8–5
interpreting return values 9–14
meaning of values 12–7
using to insert NULL values 12–6

INPUT Host Variable
definition Glossary–4

INPUT host variables 6–8
in query statements 7–3
using SQLDA for 10–22
using SQLDA to supply values 10–16

INPUT SQLDA
definition Glossary–4

INSERT statement 6–2 to 6–4
  inserting a single row 6–2
  inserting multiple rows 6–4

INTEGER data type 13–4

Integrity Constraint
  definition Glossary–4

Integrity constraints 5–9 to 5–24
  need for 5–9
  types of 5–10
    candidate key constraint 5–16
    check constraint 5–11
    primary key constraint 5–14
    referential constraint 5–18

Isolation Level
  definition Glossary–4

Italic typeface
  as typographical convention xvi

K

Keystrokes xvi

L

+L option 2–4

Linker 2–6

Locking
  EXCLUSIVE locks 11–8
  in transactions 11–7
  SHARE locks 11–8

Long-running transactions
  DDL statements in 5–24

M

Manual
  organization of xiii
  syntax notation xvii

Monospaced typeface
  as typographical convention xvi

N

NOT FOUND condition
  using the WHENEVER
    statement 9–10, 9–13

NULL keyword 8–2

Null Value
  definition Glossary–4

NULL values 8–1 to 8–10
  by default 8–3
  ignoring with aggregate
    functions 8–10
  in a GROUP BY clause 8–9
  in a WHERE clause 8–8
  in aggregate functions 8–10
  in an ORDER BY clause 8–9
  in scalar functions 8–10
  inserting into a column
    using an indicator variable 8–5
    using the NULL keyword 8–4
  retrieving
    using indicator variables 8–6
    using the NVL scalar function
      8–7
  specifying NOT NULL 8–3
  updating with 8–6
  using the DEFAULT clause to
    override 8–3

NUMERIC data type 13–5

NVL scalar function
  retrieving NULL values 8–6

O

OPEN cursor statement 7–5, 7–6, 10–6
  USING DESCRIPTOR clause 10–20

OUTPUT host variable
  definition Glossary–4

OUTPUT host variables
  in query statements 7–3
  using an SQLDA for 10–24

OUTPUT SQLDA
  definition Glossary–4

P

+P option
  ESQLC command 2–4

Parameter Marker
  definition Glossary–5

Precompiler
  definition Glossary–5
  ESQL 2–3

PREPARE statement 10–2 to 10–4
  how to use 10–3
  statements that cannot be prepared 10–3

Primary Key
  definition Glossary–5

Primary key constraint 5–14 to 5–15
  column level 5–14
  table level 5–15

Q

Query statements
  elements of 7–2
  returning a single row 7–4
  returning multiple rows 7–5

R

READ COMMITTED
  transaction isolation level 11–5

READ UNCOMMITTED
  transaction isolation level 11–5

REAL data type 13–7

Referential constraint 5–18 to 5–24
  column level foreign key 5–20
  definition Glossary–5
  table level foreign key 5–21

Referential Integrity
  definition Glossary–5

Relation
  definition Glossary–5

REPEATABLE READ
  transaction isolation level 11–6

Result Set
  definition Glossary–5

Result Table
  definition Glossary–5

ROLLBACK statement 3–13
  definition Glossary–5
  undoing changes within a transaction 11–4

S

Scalar functions
  returning NULL values 8–10
  using in SQL statements 12–7

SELECT statement 7–2
clauses 7–2
dynamic 10–6
FROM clause 7–2
INTO clause 7–2
SELECT clause 7–2
using for computations 12–4
using for condition evaluation 12–6
WHERE clause 7–2

SERIALIZABLE
transaction isolation level 11–6

SET CONNECTION statement 4–5
setting default database current 4–6
suspending a connection 4–6
using a connection name 4–5

SHARE Locks
definition Glossary–5
in transactions 11–8

SMALLINT data type 13–4

SQL
definition 1–2

SQL Communications Area
See SQLCA

SQL Descriptor area
See SQLDA

SQL reserved words 12–2

SQLCA 3–12
COMMIT statement 3–13
definition Glossary–6
for error handling 9–2
ROLLBACK statement 3–13
structure 9–2

SQLCABC 9–3

SQLCAID 9–3

SQLCODE 3–12, 9–3
definition Glossary–6
interpreting return value 9–3

sqld_alloc function 10–15, 10–25

SQLD_ARRAYSZ
setting 10–14

sqld_free function 10–19

SQLD_IVARPTRS
assigning values to 10–14

SQLD_LENGTHS 10–22

SQLD_NVARS 10–22

SQLD_TYPES 10–22

SQLD_VARPTRS
assigning values to 10–14

SQLDA 10–8 to 10–19
allocating 10–14, 10–15
components 10–10 to 10–13
SQLD_ARRAYSZ 10–13
SQLD_FLAGS 10–12
SQLD_INDVARNAMES 10–12
SQLD_INDVARNMSZ 10–12
SQLD_IVARPTRS 10–12
SQLD_LENGTHS 10–10
SQLD_NVARS 10–10
SQLD_PRECISION 10–11
SQLD_SCALE 10–12
SQLD_SIZE 10–10
SQLD_TYPES 10–11
SQLD_VARNAMES 10–12
SQLD_VARNMSIZE 10–12
SQLD_VARPTRS 10–10
definition Glossary–6
freeing 10–14, 10–19
guidelines for using 10–14
setting types and lengths 10–16
structure 10–9
using for OUTPUT host variables 10–24
when to use 10–8

SQLERRD 9–3

SQLERRM 9–3

SQLERRML 9–3

SQLERROR condition
using the WHENEVER statement 9–10

SQLERRP 9–3

SQLWARN 3–12, 9–4

SQLWARNING condition
using the WHENEVER statement 9–11

Statement String
definition Glossary–6

Static non-SELECT statements A–2

Static SELECT statements A–8

Static SQL Statement
definition Glossary–6

Static SQL statements 12–8

Substitution Name
definition Glossary–6

Syntax
ESQL command 2–3

Syntax notation xvii

T

+T option 2–4
ESQLC command 2–4, 2–6

Table
definition Glossary–6

TIMESTAMP data type 13–10

TINYINT data type 13–5

tpe_conv_data()
data conversion 13–11

TPE_DT_BINARY symbolic name 13–13, 13–18

TPE_DT_BIT symbolic name 13–13, 13–18

TPE_DT_CHAR symbolic name 13–12, 13–17

TPE_DT_DATE symbolic name 13–13, 13–17

TPE_DT_FLOAT symbolic name 13–12, 13–17

TPE_DT_INTEGER symbolic name 13–12, 13–17

TPE_DT_NUMERIC symbolic name 13–12, 13–17

TPE_DT_REAL symbolic name 13–13, 13–17

TPE_DT_SMALLINT symbolic name 13–12, 13–17

TPE_DT_TIME symbolic name 13–13, 13–17

TPE_DT_TIMESTAMP symbolic name 13–13, 13–18

TPE_DT_TINYINT symbolic name 13–12, 13–17

TPE_DT_VARBINARY symbolic name 13–13, 13–18

TPE_DT_VARCHAR symbolic name 13–12, 13–17

tpe_size
setting number of rows 7–13

Transaction
definition Glossary–6

Transaction Isolation Level
definition Glossary–6

Transaction management
statements 3–11

Transactions
abnormal termination of 11–9
COMMIT statement 11–4
forced rollback of 11–9
introduction to 11–2
isolation levels 11–5
READ COMMITTED 11–5
READ UNCOMMITTED 11–5
REPEATABLE READ 11–6
SERIALIZABLE 11–6
locking 11–7
starting 11–2

Typographical conventions xvi

U

Unique Key
definition Glossary–7

UNIQUE keyword 5–9, 5–16

UPDATE statement 6–7
updating one or more rows 6–7

V

+V option
ESQLC command 2–4

VARBINARY data type 13–11

VARCHAR data type 13–3

Variables
host 3–2
indicator 3–2

View
definition Glossary–7

W

WHENEVER statement 3–12, 9–10 to 9–13
actions
CONTINUE 9–11
GO TO 9–11
STOP 9–11
branching under exceptions 9–10
exception conditions
NOT FOUND 3–12
SQLERROR 3–12
SQLWARNING 3–12
for error handling 9–10 to 9–14
scope of 9–13
using with explicit error checking 9–13
when to use 12–2