

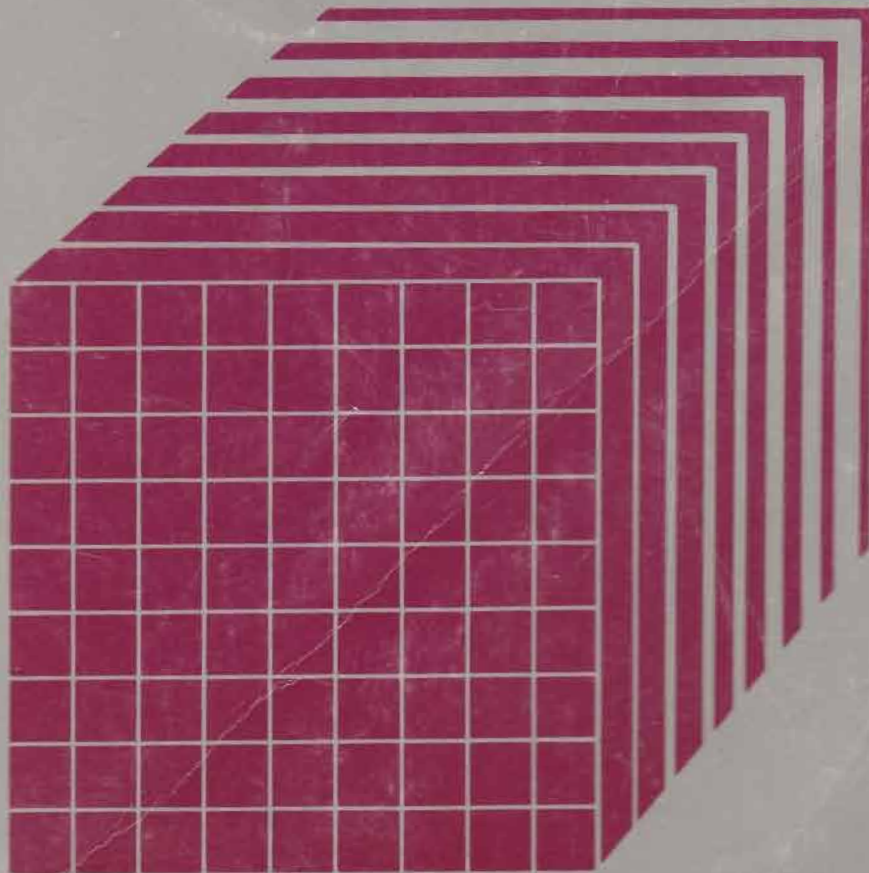


SQL/Data System Application Programming for VM/System Product

Release 3

Program Number 5748-XXJ

SH24-5068-0



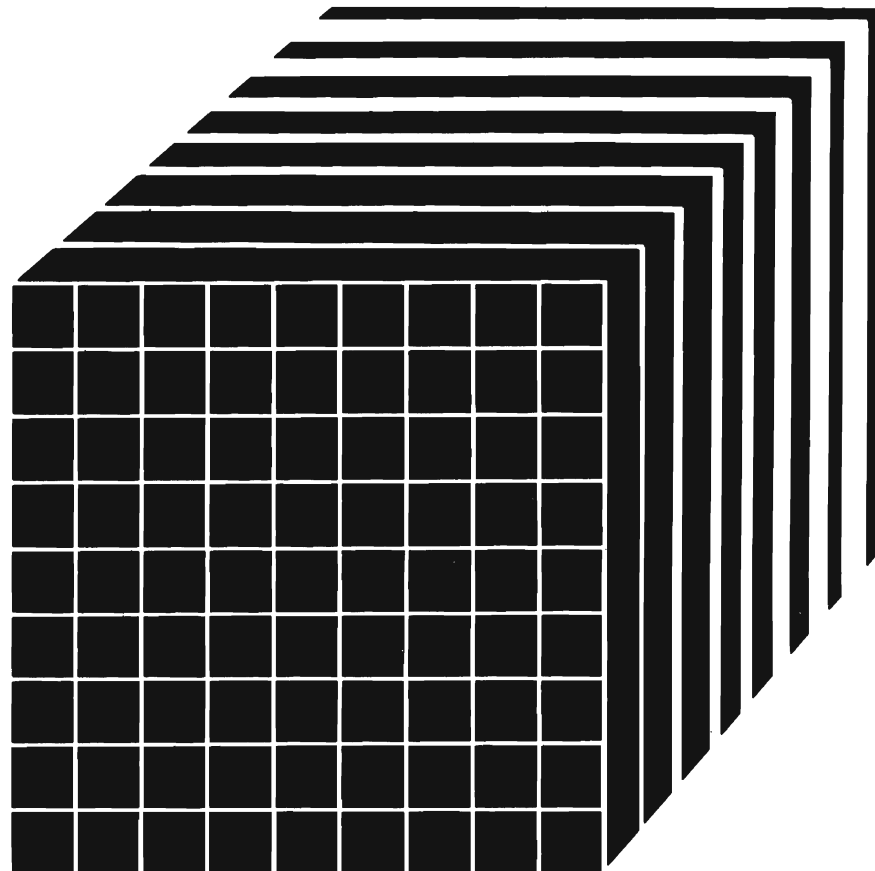


SQL/Data System Application Programming for VM/System Product

Release 3

Program Number 5748-XXJ

SH24-5068-0



First Edition (December 1984)

This edition, SH24-5068-0, is a new book based on SH24-5018-2. This edition applies to the Structured Query Language/Data System (SQL/DS) in a Virtual Machine/System Product (VM/SP) system environment. This edition applies to the Structured Query Language/Data System until otherwise indicated in new editions or Technical Newsletters. Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370 and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

Throughout this manual are illustrations in which names are used. These names are fanciful and fictitious, created by the author, and are used solely for illustrative purposes and not for the identification of any person or company.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publications, Dept. G60, P.O. Box 6, Endicott, NY, U.S.A. 13760. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Summary of Changes

This is a list of technical changes for Release 3 of SQL/DS that affect this manual. For a complete list of technical changes for Release 3, see *SQL/Data System Concepts and Facilities for VM/SP*, GH24-5065.

Performance Improvements

- Specifying the Isolation Level

Programmers can now specify whether other users can update data that the program has finished reading in its current logical unit of work. Programmers can tell SQL/DS either to lock all the data that the current logical unit of work has read, or to lock just the row or page of data that a cursor is currently pointing to.

- Dispatcher Enhancements

The SQL/DS dispatcher has been enhanced to give priority to short requests.

- FETCH and INSERT Blocking

Allows programmers to specify that a program retrieve and insert rows in groups. This can improve performance for programs running in multiple user mode which do multiple-row inserts or multiple-row SELECTs.

Enhancements for National Languages

- Specifying Character Sets

Lets an installation specify an alternative character set for national languages.

- Mixing Double-Byte Character Set (DBCS) Data and EBCDIC Data

SQL/DS Release 3 can interpret identifiers and character string constants that contain both DBCS and EBCDIC data.

Miscellaneous Enhancements

- Using Labels for Tables and Columns
Lets users define labels, which can be used as common display names, for table and column names.
- COMMENT Enhancements
Enhanced so that users can specify comments for more than one column in a single command.
- Nonrecoverable Storage Pools
Allows users to define nonrecoverable storage pools for improved performance when loading large amounts of data. With nonrecoverable storage pools, however, users must do their own data recovery.

Changes to the SQL/DS Library

- Independent Library for VM/SP users
For Release 3, separate libraries of SQL/DS manuals are available for VSE and VM/SP users.
- New Diagnostic Manuals
Two new manuals have been added to the library:
 - *SQL/Data System Diagnosis Guide for VM/SP, SY24-5230*
 - *SQL/Data System Diagnosis Reference for VM/SP, SY24-5232*These manuals help in diagnosing problems in SQL/DS. They replace the *SQL/Data System Logic* manuals.
- Technical and Editorial Changes
In addition to documenting major changes to the product, this revision incorporates minor technical and editorial changes.

Preface

This book is for application programmers writing in COBOL, PL/I, FORTRAN, or Assembler Language. It tells how to write application programs that use the Structured Query Language (SQL) to access data stored in Structured Query Language/Data System (SQL/DS) tables. Programmers writing in APL2 should refer to *APL2 Programming: Using Structured Query Language*.

Chapter 1 covers the basics of SQL programming for beginners, starting with an introduction to SQL program design. Then it explains some of the most common SQL commands. After that, there is an overview on preparing and running the program, followed by an introduction to testing and debugging concerns. The chapter ends with a section describing administrative tasks for your application program. This last section also describes the SQL/DS catalogs.

Each section in Chapter 1 has a quiz at the beginning. By taking the quiz, you can check to see how much of the material in the section you already know. Depending on your success, you may elect to skip the section after taking the quiz.

Chapter 2 expands on Chapter 1 by going into more detail and by introducing other statements in the SQL programming language. It begins by giving a detailed description of a framework for coding SQL programs. The next section explains less-common SQL commands that may be useful in coding the application. After that is a section devoted to the details of preparing and running the program. Next is a section on error handling, describing how you can use return codes set by SQL/DS to branch to error handling routines in your program. The chapter ends with a detailed description of administration considerations (including authorization, data control, and data definition).

Since Chapter 2 contains advanced information, there are no quizzes at the beginnings of the sections to check your prior knowledge.

Chapter 3 is a reference to SQL commands used in applications programming.

When you are ready to start coding, you should read about your own host language in one of the appendixes:

Appendix C, "PL/I Considerations"

Appendix D, "COBOL Considerations"

Appendix E, "Assembler Considerations"

Appendix F, "FORTRAN Considerations."

Other appendixes list SQL/DS reserved words and SQL/DS “maximums” (such as the maximum number of columns in a table or the maximum length of one SQL statement).

Also in the back of the book is a foldout. The foldout contains tables that are used in examples throughout this book.

This book assumes that you can write programs in either Assembler, PL/I, COBOL, or FORTRAN for the Virtual Machine/System Product (VM/SP). Before you read this book, you may also find it useful to know how to use the Conversational Monitor System (CMS) for VM/SP systems. *SQL/Data System Concepts and Facilities for VM/SP*, GH24-5065, is a prerequisite for this manual. You will need a copy of *SQL/Data System Messages and Codes for VM/SP*, SH24-5070; that manual explains all the return codes passed to the program by SQL/DS.

Another suggested book is the *SQL/Data System Terminal User's Guide for VM/SP*, SH24-5045, which is written in a tutorial style. It is much easier to learn SQL/DS by reading that book *before* you read this one.

Further suggested publications include:

- *SQL/Data System Installation for VM/SP*, SH24-5044
- *VM/SP CMS Command and Macro Reference*, SC19-6209
- *VM/SP CMS User's Guide*, SC19-6210
- *VM/SP CMS Primer*, SC24-5236.

Contents

- Chapter 1. Getting Started 1**
- Designing the Program 3**
- Contents 3
- Section Quiz 4
- Answers to the Section Quiz 5
- Introduction to SQL 6
- SQL Within a Programming Environment 7
- Introduction to a Framework for Coding Programs 8
- Sample Tables 9

- Coding the Program 11**
- Contents 11
- Section Quiz 12
- Answers to the Section Quiz 13
- Introduction to SQL Program Coding 14
- Retrieving One Row of Data from a Table: SELECT / INTO 14
- Retrieving or Inserting Data with a Cursor 19
- Predicates 27
- Host Variables and Constants 28
- Using Expressions as Search Conditions 30
- Built-In Functions 31
- Putting a New Row into a Table: INSERT 34
- Deleting Data from a Table: DELETE 36
- Changing Data in a Table: UPDATE 37

- Preprocessing and Running the Program 41**
- Contents 41
- Section Quiz 42
- Answers to the Section Quiz 43
- Introduction 44
- Preprocessing the Program 44
- Compiling the Program 45
- Link-Editing and Loading the Program 45
- Running the Program 45

- Testing and Debugging Concerns 47**
- Contents 47
- Section Quiz 48
- Answers to the Section Quiz 49
- Introduction 50

Using ISQL to Test SQL Statements Before Coding	50
Introduction to the SQL Communications Area (SQLCA)	51
Putting the Program into Production	53
Contents	53
Section Quiz	55
Answers to the Section Quiz	56
Authorization	57
Data Control	70
Data Definition	74
SQL/DS Catalogs	78
 Chapter 2. Advanced SQL Programming	 83
Designing the Program	85
Contents	85
Application Prolog	86
Application Body	92
Application Epilog	93
Summary	94
Sample Application Programs	95
 Coding the Program	 97
Contents	97
More About Search Conditions	99
Additions to the SELECT Statement	107
More About Cursor Management	134
More About Data Manipulation	136
Use of Views	140
Indicator Variables	146
Dynamically Defined Statements	147
 Preprocessing and Running the Program	 183
Contents	183
Introduction	184
VM/SP Connect Considerations	186
Initializing Your User Machine	186
Preprocessing the Program	187
Compiling the Program	196
Link-Editing and Loading the Program	197
Running your Program	198
 Testing and Debugging Concerns	 201
Contents	201
Error Handling	202
Monitoring Execution Performance	209
 Putting the Program into Production	 211
Contents	211
Authorization	213
Data Control	226
Data Definition	237
Performance Considerations	250

Including External Source Files	255
Including Secondary Input	255

Chapter 3. SQL Programming Language Reference Summary ... 257

How to Interpret SQL Format	259
--	------------

SQL Statement Reference Summary

Contents	261
ACQUIRE DBSPACE	263
ALTER DBSPACE	265
ALTER TABLE	267
BEGIN DECLARE SECTION	268
CLOSE	269
COMMENT	270
COMMIT WORK	272
CONNECT	273
CREATE INDEX	274
CREATE SYNONYM	276
CREATE TABLE	277
CREATE VIEW	279
DECLARE CURSOR	281
DELETE	284
DESCRIBE	286
DROP DBSPACE	288
DROP INDEX	289
DROP PROGRAM	290
DROP SYNONYM	291
DROP TABLE	292
DROP VIEW	293
END DECLARE SECTION	294
EXECUTE	295
EXECUTE IMMEDIATE	296
EXPLAIN	297
FETCH	299
GRANT	300
INSERT	305
LABEL	309
LOCK	312
OPEN	313
PREPARE	314
PUT	316
REVOKE	317
ROLLBACK WORK	321
SELECT	322
UPDATE	324
UPDATE STATISTICS	328
WHENEVER	329

Chapter 4. Extended Dynamic Statements

Contents	331
Purpose and Use of Extended Dynamic Statements	332

An Example of Extended Dynamic Statements	336
Logical Unit of Work Considerations	344
Extended Dynamic Statement Descriptions	348
Appendix A. SQL/DS Reserved Words	363
Appendix B. SQL/DS Maximums	365
Appendix C. PL/I Considerations	367
ARISPLIC -- PL/I Sample Program	367
Rules for Using SQL in PL/I	375
SQL Error Handling	380
Dynamic SQL Statements in PL/I	381
Data Types	383
Additional PL/I Program Examples	383
Appendix D. COBOL Considerations	397
ARISCBLC -- COBOL Sample Program	397
Rules for Using SQL in COBOL	410
SQL Error Handling	415
Dynamic SQL Statements in COBOL	416
Data Types	417
Additional COBOL Program Example	419
Appendix E. Assembler Considerations	423
Acquiring the SQLDSECT Area	423
Performance Considerations for the SQLDSECT Area	424
ARISASMC -- Assembler Sample Program	426
Rules for Using SQL in Assembler	441
SQL Error Handling	444
Dynamic SQL Statements in Assembler	445
Data Types	446
Reentrant Programs	447
Appendix F. FORTRAN Considerations	453
ARISFTN -- FORTRAN Sample Program	453
Rules for Using SQL in FORTRAN	461
SQL Error Handling	465
Dynamic SQL Statements in FORTRAN	466
Data Types	466
Index	469

Figures

1.	INVENTORY Table	6
2.	Form of Embedded SQL Statements	14
3.	Format of the SELECT Statement	15
4.	Using a Cursor	21
5.	Breakdown of Search Conditions and Predicates	27
6.	Breakdown of an Expression	30
7.	Hierarchy of SQL/DS Authority	61
8.	Privileges You Can Grant	63
9.	Locking Summary for PRIVATE DBSPACES	71
10.	Locking Summary for PUBLIC DBSPACES	72
11.	SQL/DS Data Types	76
12.	SQL/DS Data Conversion Chart	77
13.	Examples of Host Variable Declarations	88
14.	Examples of Embedded SQL Statements	90
15.	SQL Declarative Statements	92
16.	Pseudo Code Framework for Coding Programs	95
17.	Truth Table for Null Values	102
18.	Values Returned in Indicator Variables	146
19.	SQLDA Structure (in Pseudo Code)	152
20.	Data Codes Returned in SQLTYPE	155
21.	SQLDA Initialization	167
22.	Using a Cursor with Dynamically Defined Statements	179
23.	SQL/DS Modes of Operation	185
24.	SQLCA Structure (in Pseudo Code)	202
25.	Pseudo-Code Error Handling Routine	208
26.	Default Table Placement	240
27.	Variable Names for Specifying Mixed Isolation Levels	252
28.	Relationship Between Extended Dynamic Statements Expressed Using Host Program Variables	334
29.	Dynamic vs. Extended Dynamic Statements	335
30.	An Example of an Interpretive Support Program for Building and Executing SQL Statements in an Access Module	337
31.	Preprocessing and Assembly of a Two-Part Support Program	340
32.	Preprocessing and Execution of an End-User Program by a Two-Part Support Program	341
33.	Pseudo-Code Example of Preprocessing End-User Program P	343
34.	Pseudo-Code Example of Execution of End-User Program P	344
35.	Placement of Extended Dynamic Statements in Logical Units of Work	346
36.	Ranges of Numeric Values	365
37.	SQLCA Structure (in PL/I)	380
38.	SQLDA Structure (in PL/I)	381
39.	SQLDAX Structure (in PL/I)	381
40.	SQL/DS Data Types for PL/I	383

41.	SQLCA Structure (in COBOL)	416
42.	SQL/DS Data Types for COBOL	417
43.	Acquiring the SQLDSECT Area for VM/SP Applications	423
44.	SQLCA Structure (in Assembler)	444
45.	SQLDA Structure (in Assembler)	445
46.	SQL/DS Data Types for Assembler	446
47.	SQL Statements Supported in FORTRAN	461
48.	SQLCA Structure (in FORTRAN)	466
49.	SQL/DS Data Types for FORTRAN	466

Chapter 1. Getting Started

This chapter teaches you the basics of how to develop an application program in Structured Query Language/Data System (SQL/DS). It is divided according to the tasks that you, as an application programmer, are likely to perform. Chapter 2 contains advanced information on each of these tasks. If you need to know more about a particular task, you can just refer to the advanced version of your section in Chapter 2. For instance, if you are reading about designing your program, and you would like to know more about designing before you begin reading about coding your program, you can skip to the advanced version of Designing the Program in Chapter 2.

Each of the sections in this chapter is preceded by a section quiz. The quizzes are to help you determine how much of the information in the section you already know. If you think the quiz is easy, skip that section and proceed to the next one. Because the material in Chapter 2 is more advanced, Chapter 2 does not have any section quizzes.

Designing the Program

This section begins with an introduction to SQL. It goes on to discuss SQL within a programming environment, and concludes with an introduction to the form that SQL programs usually take.

Contents

Section Quiz	4
Answers to the Section Quiz	5
Introduction to SQL	6
SQL Within a Programming Environment	7
Introduction to a Framework for Coding Programs	8
Sample Tables	9

Section Quiz

If you can answer most or all the following questions, then you probably do not have to read this section. You could browse through the section for review, or you could skip to “Coding the Program” on page 11. If you have trouble answering the questions in this quiz, proceed to “Introduction to SQL” on the next page.

1. What is the form that SQL/DS data takes?
2. SQL commands can be embedded in host language programs written in:
 - a. _____
 - b. _____
 - c. _____
 - d. _____

(Four different programming languages).
3. What are the five steps or tasks to developing an application program?
4. What are the three steps you must do to prepare your program before running it?
5. What must you place in the application prolog?
6. What needs to be coded in the application epilog?
7. What part of the program contains the SQL statements?

Answers to the Section Quiz

1. Tables
2. FORTRAN, COBOL, PL/I, and Assembler language
3. 1. Designing, 2. Coding, 3. Preparing and Running, 4. Error Handling, 5. Administrating
4. 1. Preprocess, 2. Compile, 3. Load
5. Statements that provide for error handling, declare host variables, and establish a connection between your program and SQL/DS
6. Statements that tell SQL/DS what to do with changes made to data, and release the program's connection to SQL/DS
7. The application body.

Introduction to SQL

The Structured Query Language/Data System (SQL/DS) is a data base management system that uses the relational data model of data. You can think of a relational data model as being a collection of tables where a **relation** in this model is one of these tables. A table in the relational data model is no different than any other simple two-dimensional table. It has a specific number of columns and some number of unordered rows, and a specific item of data at the intersection of every column and row. Data is accessed in terms of tables and operations on tables. That is, you can get SQL/DS data just by knowing the names of the table and the column that it is in. This provides for an easy-to-use set of commands which let you work with the data, without having to bother with the way in which the data is actually stored in the system.

Look at the sample tables in the foldout in the back of the book. These are examples of tables in SQL/DS. We will be referring to them in examples throughout the book. The INVENTORY table, shown in Figure 1, has columns named PARTNO, DESCRIPTION, and QONHAND.

PARTNO	DESCRIPTION	QONHAND
207	GEAR	75
209	CAM	50
221	BOLT	650
222	BOLT	1250
231	NUT	700
232	NUT	1100
241	WASHER	6000
285	WHEEL	350
295	BELT	85

Figure 1. INVENTORY Table

Suppose, for example, you wanted to get a listing of all the different part names (descriptions) of the parts that your company stocked. You could get this data simply by knowing the name of the table, INVENTORY, and the name of the column, DESCRIPTION, that the data was in. Then you would code this in an appropriate SQL statement.

The language for handling SQL/DS data is called the Structured Query Language (SQL). This language contains commands that retrieve, delete, and update tables in the SQL/DS data base. You can embed these commands in application programs written in COBOL, FORTRAN, PL/I, or Assembler Language. These commands do all the data handling on SQL/DS data. With them, you use the power of SQL/DS and decrease the data handling done by the programs themselves. Programs that access SQL/DS data can also access data from other sources, such as CMS files.

You can use SQL/DS under the Virtual Machine/System Product (VM/SP) operating system. Application programs can be:

- Online programs operating in virtual machines. These programs are controlled by the Conversational Monitor System (CMS).
- Non-interactive programs operating in virtual machines in VM/SP.

SQL Within a Programming Environment

Programs that use SQL/DS can be written in COBOL, PL/I, FORTRAN, or Assembler Language. These languages are called host languages because they act as hosts for SQL. Application programs work with SQL/DS data through SQL statements that you embed in the programs. How you embed SQL statements varies slightly for each of the four languages that SQL/DS supports.

The core of SQL is the same for each of the host languages. For this reason, SQL is presented throughout this book in *basic form*, unless otherwise noted. That is, the SQL statements are shown without any of the language-dependent delimiters. The SQL syntax and examples in this book are language independent.

Examples that have combinations of SQL statements and host language statements are also shown in a language-independent form called *pseudo code*. Pseudo code shows program logic but must be re-coded in a specific programming language before it can be used. When SQL statements are shown in pseudo code examples, they are preceded by EXEC SQL to help you distinguish the SQL statements from the pseudo code. When shown by themselves, SQL statements are not preceded by EXEC SQL.

For you to use SQL in a particular programming language, you must be familiar with the rules for embedding SQL statements in that language. These language rules are discussed in the appendixes; each programming language has a separate appendix devoted to it:

Appendix C, "PL/I Considerations."

Appendix D, "COBOL Considerations."

Appendix E, "Assembler Considerations."

Appendix F, "FORTRAN Considerations."

You should glance over the appendix of the programming language that you will be using before you continue reading. Don't worry right now about understanding the SQL statements coded in the example programs in the appendixes. Just try to get a general feel for how the statements are embedded. The SQL language is explained in the first two chapters of this book. Once you are ready to code your first SQL/DS application, you will probably need to refer to the appendixes again to help you code. At that point, you can refer to the third chapter of this book, "Chapter 3. SQL Programming Language Reference Summary" on page 257 for reference information on each of the SQL statements that you learned.

Writing your program consists of a series of steps or tasks. This book is organized according to these tasks.

1. The first step is designing your program, or determining what you want the program to do. This also includes choosing the type of application and developing the structure or framework of the program. This information is covered in the “Designing the Program” sections of both Chapters 1 and 2. Chapter 1 contains basic information; Chapter 2 is for advanced programmers.
2. The second step is coding your SQL program. This consists of using the SQL statements and tools to access and work with SQL/DS data, according to the purpose set forth in your design. The second section of both Chapters 1 and 2 presents the various SQL statements and tells you how to use them.
3. The third step in developing your program is to get it ready to be run. This includes:
 - a. Preprocessing the SQL code using one of the SQL/DS preprocessors
 - b. Compiling the code using the compiler of your host language to produce an object program
 - c. Loading the object program to be run
 - d. Executing the program to carry out operations on the tables.

The “Preprocessing and Running the Program” sections of both Chapters 1 and 2 tell you how to get your program ready to be run.

4. The fourth step is to debug the program. This involves testing for errors that may become apparent during preprocessing, compiling, loading or during execution. The “Testing and Debugging Concerns” sections help you with this task.
5. Finally, you must act as an administrator. You must control who can run your program and who can use the data that your program accesses. You may have to create your own data. You may need information about your data tables, such as who first created them, or who else can access them. All this information is in the “Putting the Program into Production” sections.

Also, once you have mastered the SQL language, you will probably need reminders on the syntax or parameters of an SQL statement. The third chapter of this book is written for this purpose.

Introduction to a Framework for Coding Programs

You can think of an SQL/DS application program as containing three main parts: the prolog, the body and the epilog. You must place certain SQL statements at the beginning and end of the program to handle the transition from the host language to the embedded SQL statements. For instance, your program must establish a connection to SQL/DS before the SQL statements can access data. Similarly, your program must also release this connection after it is done using the data base. Also, every SQL/DS application must provide for error handling. Statements to do these things are put in the application prolog and the application epilog.

The application prolog should be at the beginning of every SQL program. In the prolog, you must place SQL statements that do the following:

- Provide for error handling by setting up a communications area.
- Declare special variables (host variables) that SQL/DS uses to interact with the host program. Host variables are really just normal host program variables that are used in SQL statements. The only difference is that when they are coded in an SQL statement, these variables must be preceded by a colon (:). But they work just like regular program variables.
- Establish a connection between your program and SQL/DS.

The statements that do these things are described in “Designing the Program,” Chapter 2.

The application body is where you place the SQL statements that operate on SQL/DS tables. These statements are covered in the “Coding the Program” sections of this book.

The application epilog is at the end of every SQL application program. It must contain SQL statements that:

- Tell SQL/DS what to do with changes made to data. Changes can either be saved (“committed”) or ignored (“rolled back”).
- Release the program’s connection to SQL/DS.

Again, the statements that do these things are detailed in the advanced version of this section, “Designing the Program,” Chapter 2.

Sample Tables

The foldout at the end of the book contains a set of tables. These tables are used in an inventory control application for a small manufacturing company. They are used throughout the book for SQL statement examples.

The INVENTORY table lists the part number, description, and quantity on hand of each part in the inventory. The SUPPLIERS table lists the supplier number, name, and address of the various companies that supply parts. The QUOTATIONS table lists the part numbers that can be obtained from each supplier, together with the current price and delivery time (in days) promised by the supplier for the given part. The QUOTATIONS table also lists the quantity on order for each part from a given supplier.

Next to the INVENTORY table is a list of all the columns with their corresponding SQL/DS data types. Examples in subsequent chapters refer to these data types.

Coding the Program

This section tells you how to code data retrieval (SELECT) statements and data manipulation (INSERT, DELETE, and UPDATE) statements in SQL. This section also shows you some other things that you can use in SQL statements, such as constants, host variables, and built-in functions.

Contents

Section Quiz	12
Answers to the Section Quiz	13
Introduction to SQL Program Coding	14
Retrieving One Row of Data from a Table: SELECT / INTO	14
SELECT Clause: Expressing Desired Results	15
INTO Clause: Returning a Single Row	17
FROM Clause: Specifying a Table Name	18
WHERE Clause: Searching on Conditions	19
Retrieving or Inserting Data with a Cursor	19
DECLARE CURSOR Statement	22
OPEN Statement	23
FETCH Statement	23
PUT Statement	25
CLOSE Statement	26
Predicates	27
Host Variables and Constants	28
Using Expressions as Search Conditions	30
Built-In Functions	31
Putting a New Row into a Table: INSERT	34
Deleting Data from a Table: DELETE	36
Changing Data in a Table: UPDATE	37

Section Quiz

The following questions cover the high points of “Coding the Program.” If you can answer all of these questions, you probably do not need to read this section. If you decide to skip ahead, proceed to “Preprocessing and Running the Program” on page 41. If you have trouble answering these questions, you should read the whole section.

1. Write an SQL statement that would retrieve the part number and price from the QUOTATIONS table for all the parts supplied by supplier number 54. You should use the DECLARE CURSOR format since more than one row will be returned. Choose your own cursor name. (Refer to the sample tables in the foldout in the back of the book.)
2. Write an SQL statement that would retrieve the part number, price and delivery time from the QUOTATIONS table where the supplier number is 53 and the part number is 232. Unless you are a FORTRAN coder, you should use the SELECT / INTO format with this one, since only one row will be returned. If you use FORTRAN, you should answer with the DECLARE cursor format. Make up your own names for the host variables or cursor.
3. Suppose you just coded the following DECLARE CURSOR statement:

```
DECLARE CCC CURSOR FOR
SELECT SUPPNO, PARTNO
FROM QUOTATIONS
WHERE PRICE > 10.00
ORDER BY PRICE
```

Write three statements that will first open the cursor, then put the first row of the active set into host variables SUPP and PART, and then close the cursor once again. You do not need to begin your statements with EXEC SQL for this exercise.

4. Write an SQL statement that will find the amount you will have to pay supplier number 53 when part number 222 arrives. Include a 7% sales tax.
5. Write a statement that finds the difference between the maximum and the minimum price for part number 221. Name the cursor anything you like.
6. Write an SQL statement that would delete all the rows from the INVENTORY table that have a part number greater than 270.
7. Write a statement that would put a new row into the INVENTORY table. Let the new row describe a part number 252 that has a description of LEVER and a quantity on hand of 25.
8. Write a statement that would change the address of SKY PARTS to 310 SATURN ST., MILKYWAY NY.

Answers to the Section Quiz

1. DECLARE C1 CURSOR FOR
SELECT PARTNO, PRICE
FROM QUOTATIONS
WHERE SUPPNO = 54

2. SELECT PARTNO, PRICE, DELIVERY_TIME
INTO :PART, :PRI, :DEL
FROM QUOTATIONS
WHERE SUPPNO=53 AND PARTNO=232

or

DECLARE C2 CURSOR FOR
SELECT PARTNO, PRICE, DELIVERY_TIME
FROM QUOTATIONS
WHERE SUPPNO=53 AND PARTNO=232

3. OPEN CCC
FETCH CCC INTO :SUPP, :PART
CLOSE CCC

4. SELECT QONORDER*PRICE + QONORDER*PRICE*.07
INTO :COST
FROM QUOTATIONS
WHERE SUPPNO=53 AND PARTNO=222

or

DECLARE C3 CURSOR FOR
SELECT QONORDER*PRICE + QONORDER*PRICE*.07
FROM QUOTATIONS
WHERE SUPPNO=53 AND PARTNO=222

5. SELECT MAX(PRICE)-MIN(PRICE)
INTO :RANGE
FROM QUOTATIONS
WHERE PARTNO=221

or

DECLARE C4 CURSOR FOR
SELECT MAX(PRICE)-MIN(PRICE)
FROM QUOTATIONS
WHERE PARTNO=221

6. DELETE FROM INVENTORY
WHERE PARTNO > 270

7. INSERT INTO INVENTORY
VALUES (252, 'LEVER', 25)

8. UPDATE SUPPLIERS
SET ADDRESS = '310 SATURN ST., MILKYWAY NY'
WHERE NAME = 'SKY PARTS'

Introduction to SQL Program Coding

Application programmers using SQL/DS have some very useful statements for retrieving and manipulating data at their disposal. As mentioned earlier, the coding task in the SQL/DS application program consists of embedding these statements into the host language code. The delimiters for SQL statements differ for each host language. In each language, SQL statements are prefixed by "EXEC SQL". But, in COBOL, the end of the command is denoted by "END-EXEC", while in PL/I, the usual semi-colon (;) is used. There is no trailing delimiter for *Assembler* or FORTRAN. Examples of the general form of embedded SQL commands are shown in the following chart for each host language:

Host Language	Format of Embedded Statement
COBOL	EXEC SQL sql-statement END-EXEC
PL/I	EXEC SQL sql-statement;
Assembler	EXEC SQL sql-statement
FORTRAN	EXEC SQL sql-statement

Figure 2. Form of Embedded SQL Statements

The exact rules of placement, continuation and delimiting of SQL statements are in the host language appendixes. This section, "Coding the SQL Program," contains explanations of how to code SQL/DS data retrieval and manipulation commands for one or more rows of data from an SQL/DS table, without going into the details of the different host languages.

One of the most common things an SQL application programmer must do is **retrieve data** from the data base. In SQL, this is achieved through the use of the SELECT statement. The SELECT statement is a form of query. It searches the SQL/DS data base to see if any rows of any tables in the data base meet search conditions specified in the SELECT statement. If any such rows exist, the data is retrieved from the data base and put into specified variables in the host program. Then the program can use this data for whatever it was designed to do.

There are two types of SELECT statements. The first, the SELECT/INTO version, is used to retrieve only a single row of data from the data base. The second, the DECLARE cursor version, is used to retrieve more than one row of data. The SELECT/INTO statement cannot be used in FORTRAN programs. Instead, for FORTRAN, the DECLARE cursor version is used to retrieve one or more rows of data.

Retrieving One Row of Data from a Table: SELECT / INTO

The SELECT statement for retrieving one row of data is made up of four clauses: the SELECT clause, the INTO clause, the FROM clause and the WHERE clause. You must specify the clauses in that order.

```
SELECT select-list
INTO one-or-more-host-variables
FROM table-name
[ WHERE search-condition ]
```

Figure 3. Format of the SELECT Statement

Note: Remember that the SELECT / INTO statement cannot be used in FORTRAN programs. Instead, you must use a cursor to retrieve data from one or more rows in a table. Cursors (the DECLARE CURSOR statement) are described under “Retrieving or Inserting Data with a Cursor” on page 19.

The SELECT / INTO statement finds one row of the table specified in the FROM clause that satisfies the given search condition. From this row SQL/DS selects the columns that you have supplied in the select-list. The results are delivered into the host variables that you have listed in the INTO clause. For example, the following statement *selects* the part number, description, and quantity on hand *from* the INVENTORY table *where* the description of the part is ‘BOLT’. It places the result *into* the host variables PART, DESC, and QUANT:

```
SELECT PARTNO, DESCRIPTION, QONHAND
INTO :PART, :DESC, :QUANT
FROM INVENTORY
WHERE DESCRIPTION = 'BOLT'
```

The SELECT, INTO and FROM clauses are required for every SELECT statement that you code. The WHERE clause is the only one of the four that is optional. If you do not supply a WHERE clause, all rows of the table qualify.

Now let’s take a look at each of the four SELECT statement clauses in more detail.

SELECT Clause: Expressing Desired Results

```
SELECT select-list
INTO one-or-more-host-variables
FROM table-name
[ WHERE search-condition ]
```

The SELECT clause is the first part of a SELECT statement. It consists of the keyword SELECT followed by a *select-list*.

The select-list is made up of one or more column names or expressions separated by commas. (Expressions will be explained in detail under section “Using Expressions as Search Conditions” on page 30.)

The following are examples of select-lists that might occur in queries to the example tables in the foldout:

```

SELECT DESCRIPTION, QONHAND
SELECT QONHAND - :X, PARTNO
SELECT DELIVERY_TIME + 10
SELECT 250
SELECT PRICE * .85

```

If you specify **DISTINCT** immediately after the word **SELECT**, SQL/DS eliminates duplicates from the query-result. (You can use **DISTINCT** only once in any query.) For example, the **SELECT** clause below returns the set of different supplier numbers in the rows that satisfy the search condition.

```

SELECT DISTINCT SUPPNO

```

SUPPNO
55
55
56
57

SQL/DS returns only one of these.

Similarly, the following **SELECT** clause returns the set of different pairs of supplier numbers and part numbers from rows that satisfy the search condition.

```

SELECT DISTINCT SUPPNO, PARTNO

```

SUPPNO	PARTNO
55	206
55	207
55	207
55	208

SQL/DS returns only one of these.

ALL indicates that duplicates are not to be eliminated, and is the default.

SQL provides a special shorthand notation for selecting all the fields of a row:

```

SELECT *

```

For example, the following statement returns the entire row from the **SUPPLIERS** table for supplier number 51:

```

SELECT *
INTO :SUPPNO, :NAME, :ADDR
FROM SUPPLIERS WHERE SUPPNO=51

```

If you specify a constant as a select-list expression, that constant occurs in every row returned by the query. For example, the following figure shows a query that returns a constant and all the supplier names.

```
SELECT 'NAME IS', NAME
FROM SUPPLIERS
```

EXPRESSION 1	NAME
NAME IS	DEFECTO PARTS
NAME IS	VESUVIUS, INC.
NAME IS	ATLANTIS CO.
NAME IS	TITANIC PARTS
NAME IS	EAGLE HARDWARE
NAME IS	SKY PARTS
NAME IS	KNIGHT LTD.

Note: Remember that the SELECT/INTO version of the SELECT statement can only be used to retrieve a single row of data from the data base. Since the above statement returns more than one row, you would have to declare a cursor to retrieve these rows to your program. Cursors are discussed later in this section.

Note the difference between constants and SQL identifiers in select-lists. An alphabetic constant (such as 'NAME IS' in the above example) is *always* enclosed within single quotes (') when used in an SQL statement. A numeric constant does not have to be enclosed within single quotes. An SQL identifier must be enclosed within double quotes (") when it contains blanks or special symbols (such as "SALARY COLUMN"). See "General Rules for Naming Data Objects" on page 74 for additional information.

INTO Clause: Returning a Single Row

```
SELECT select-list
INTO one-or-more-host-variables
FROM table-name
[ WHERE search-condition ]
```

Note: The INTO clause is not supported for FORTRAN. Instead, you must use a cursor to fetch the row.

You can think of the result of a SELECT statement as a table having rows and columns, much like a table in the data base. If the SELECT statement returns only one row, SQL/DS delivers the results directly into the host variables specified in the INTO clause. If the SELECT statement returns more than one row, you must use a *cursor* to fetch the rows one at a time. Cursors are described under "Retrieving or Inserting Data with a Cursor" on page 19. For the remainder of this section, assume that the SELECT statement returns a single row. The following are several examples of this type of SELECT statement:

```
SELECT ADDRESS
INTO :X
FROM SUPPLIERS
WHERE NAME='EAGLE HARDWARE'
```

```

SELECT QONHAND + 100
INTO :Q
FROM INVENTORY
WHERE PARTNO = 221

```

```

SELECT PRICE, DELIVERY_TIME
INTO :P, :D
FROM QUOTATIONS
WHERE SUPPNO=:X AND PARTNO=:Y

```

If the number of expressions in the select-list is not equal to the number of main host variables provided in the INTO clause, a warning flag (called SQLWARN3) in the SQLCA is set to 'W'. (See "Error Handling" on page 202 for a description of the SQLCA.) The number of values returned is either the number of expressions in the select-list or the number of host variables in the INTO clause (whichever is smaller).

The host variables in the INTO clause must be compatible with the expressions in the select-list. Integers, small integers, decimal numbers, and floating point numbers are compatible; fixed-length, varying-length, and LONG VARCHAR character strings are compatible; and fixed-length, varying-length, and LONG VARGRAPHIC DBCS character strings are compatible.

Just before delivering each selected item into its associated host variable, SQL/DS converts the selected item (if necessary) to the data type of the host variable. Conversion from decimal or floating point to integer is done by truncation (for example, 2.75 is truncated to 2). Whenever a floating point number is converted to decimal, the decimal number acquires a precision of 15 and the maximum scale that allows the integer part of the number to be represented without loss of either significance or accuracy. Any necessary truncation is toward zero; that is, all losses are on the right.

If the data value is too large for successful conversion (as when a decimal value is larger than the largest representable integer), SQL/DS indicates a conversion error by returning a negative SQLCODE in the SQLCA. When a conversion error occurs, the contents of the host variable are unpredictable. Thus, if you are doing your own error handling, you should always check SQLCODE after executing a SELECT statement. When a decimal number is assigned to a decimal variable, the number is converted, if necessary, to the precision and scale of the target. SQL/DS data conversion is summarized under "Data Conversion" on page 76.

FROM Clause: Specifying a Table Name

```

SELECT select-list
INTO one-or-more-host-variables
FROM table-name
[ WHERE search-condition ]

```

Use the FROM clause to specify the name of the table from which you want to retrieve data. If the table is owned by another user, you can access it, if you are so authorized, by concatenating with a period the userid of the owner of the table to

the table-name itself. For example, to specify table SUPPLIERS owned by user SMITH:

```
FROM SMITH.SUPPLIERS
```

Because any number of users can define a table with the same name, it is strongly recommended that you always use fully qualified table names. This avoids confusion if you are writing a program that someone else will preprocess.

WHERE Clause: Searching on Conditions

```
SELECT select-list  
INTO one-or-more-host-variables  
FROM table-name  
[ WHERE search-condition ]
```

The WHERE clause is the place to specify your search conditions. If you don't specify a WHERE clause, all the rows of the table are used to compute the expressions in the select-list. Here are some examples of WHERE clauses:

```
WHERE ITEM = :X  
  
WHERE QONORDER < :R1  
AND :FLAG = 0
```

If more than one row satisfies the search condition in a SELECT / INTO statement, an error condition occurs and no rows are returned.

For more information on search conditions, see "More About Search Conditions" on page 99.

Retrieving or Inserting Data with a Cursor

The previous section showed how to use a SELECT statement in COBOL, PL/I, and Assembler language programs to retrieve certain fields from a single row of a table. In FORTRAN programs, an SQL/DS *cursor* must be used to retrieve data from one or more rows of a table. In COBOL, PL/I and Assembler language, cursors must be used for queries that could return more than one row of a table.

A cursor, in general terms, is a pointer to the data base. SQL/DS cursors should not be confused with the cursors found on display terminals.

The SQL DECLARE CURSOR statement defines a cursor by associating a name of your own choosing with a query. The query may return many rows from the data base. The rows of the result are called the *active set* of the cursor.

A cursor can be in an *open* state or a *closed* state. When the cursor is in the open state, it maintains a position in its active set in one of three places:

1. On a certain row (called the *current row*);
2. Between two rows; or
3. Before the first row.

Once you have defined a cursor, you can manipulate it using the following statements:

OPEN If the cursor is a query-cursor (a cursor defined in terms of a *SELECT* statement), the *OPEN* statement examines the contents of the host variables, if any, in the *WHERE* clause of the query associated with the cursor. The host variables of the *WHERE* clause are called *input host variables* because they furnish information needed in the processing of the query. By evaluating the input host variables, the *OPEN* statement determines the set of rows that satisfy the query.

The cursor is placed in the open state and its active set becomes the set of rows that satisfy the *WHERE* clause. However, none of these rows is actually retrieved from the data base yet (this is done by *FETCH*). The cursor is placed just before the first row of the active set. Once you have opened a cursor, the input host variables are not re-examined (and hence no change occurs to the contents of the active set) until you close and re-open the cursor.

If the cursor is defined in terms of an *INSERT* statement, and your program is blocking, the *OPEN* statement tells SQL/DS to prepare to block. (Blocking is discussed in detail under "To Block or Not to Block?" on page 254.) Even if your program is not blocking, you should still *OPEN* and *CLOSE* every cursor, including insert-cursors.

FETCH Advances the position of the cursor to the next row of its active set and delivers the selected fields of that row into the *output host variables*. The output host variables are designated in the *FETCH* statement. (No *INTO* clause is used in the *SELECT* statement associated with the cursor.) When there are no rows remaining to be fetched in the active set, SQL/DS returns the "not found" result code (*SQLCODE=100*).

PUT If your program is blocking, the *PUT* statement inserts the contents of the input host variables into the insert-block. The input host variables are defined in the *VALUES* clause of the *DECLARE CURSOR* statement. After the *PUT* statement is executed, you can redefine the input host variables to add another row to the insert-block. Rows are not inserted into the data base until the block is full, or until a *CLOSE* statement is issued. If blocking is not in effect, the *PUT* statement simply inserts one row of data directly into a table as determined by the insert-cursor.

DELETE Deletes one row of a table determined by the current position of the cursor in the active set. This statement does not directly affect the position of the cursor, but because the row it was positioned on is deleted, the cursor is left in the *between* position. The cursor cannot

be used for further deletions or updates until it is repositioned by a FETCH statement.

UPDATE Updates one row of a table determined by the current position of the cursor in the active set. This statement does not affect the position of the cursor.

CLOSE Closes the cursor; the active set of the cursor becomes undefined. No FETCH or PUT statements can be issued against the cursor until it is re-opened. Note that both the COMMIT WORK and ROLLBACK work statements automatically close all cursors. It is recommended, however, that you always explicitly close all cursors when they are no longer needed.

Figure 4, which is a pseudo code program fragment, illustrates the use of a cursor named C1. C1 finds the part numbers and prices of all the rows of the QUOTATIONS table whose supplier number matches host variable SUPP. FETCH statements retrieve the selected fields successively into host variables PART and PRICE. Once retrieved, the results are displayed on the console.

```
SUPP = 51          <---- Initialize SUPP (the input host variable).
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT PARTNO,PRICE
  FROM QUOTATIONS WHERE SUPPNO=:SUPP
  ORDER BY PARTNO
]----- Declare cursor
]----- Open the cursor.
EXEC SQL OPEN C1
EXEC SQL FETCH C1 INTO :PART, :PRICE
DO WHILE (SQLCODE=0)
  DISPLAY (PART, PRICE)
  EXEC SQL FETCH C1 INTO :PART,:PRICE <-- host variables and display
  them.
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE C1 <--- When the active set is empty,
                    close the cursor.
```

Figure 4. Using a Cursor

Recall that SQLCODE is set to 100 when there are no rows remaining to be fetched.

The formats of the statements for cursor management, and the details of their use, are described below.

DECLARE CURSOR Statement

Format 1:

```
DECLARE cursor-name CURSOR FOR select-statement
    [ ORDER BY o-spec [ASC|DESC] [, o-spec [ASC|DESC] ] ...
    [ FOR UPDATE OF column-name-1 [, column-name-2 ] ... ]
```

Format 2:

```
DECLARE cursor-name CURSOR FOR insert-statement
```

Example (Format 1):

```
DECLARE C1 CURSOR FOR SELECT PARTNO, PRICE
    FROM QUOTATIONS WHERE SUPPNO=:SUPP
    ORDER BY PARTNO
```

Example (Format 2):

```
DECLARE C2 CURSOR FOR INSERT INTO INVENTORY
    (PARTNO, DESCRIPTION, QONHAND)
    VALUES (:PART, :DESC, :QUAN)
```

Authorization:

Anyone connected to SQL/DS can issue this statement. You must, however, be authorized to access the tables referenced in the SELECT or INSERT statement. (See the statement authorization descriptions for these statements.)

This statement defines a cursor by associating a *cursor-name* with the specified *select-statement* or *insert-statement*. Cursor names must be unique in a logical unit of work. The DECLARE CURSOR statement should not be confused with a host variable declaration. The SQL DECLARE CURSOR statement should never be placed within the host variable declare section.

A cursor-name must begin with a letter, \$, #, or @. It can contain up to 18 letters, \$, #, @, underscores, and numbers. Unlike other SQL identifiers, cursor names must never be enclosed in either single (') or double (") quotes; thus, cursor names cannot contain embedded blanks. Cursor names can, however, be SQL reserved words. For example:

```
DECLARE DELETE CURSOR FOR SELECT PARTNO FROM INVENTORY
```

Note that the cursor name above (DELETE) is *not* enclosed in double quotes. If you refer to such a cursor name in an UPDATE statement, however, you must enclose the cursor name in double quotes. (See Format 2 of the UPDATE statement in the next chapter.)

The select-statement or insert-statement is actually a part of the DECLARE CURSOR statement, so you must not precede SELECT or INSERT with EXEC SQL. (However, EXEC SQL is placed in front of the DECLARE.)

OPEN Statement

Format:

```
OPEN cursor-name
```

Example:

```
OPEN C1
```

If you are opening a query-cursor, this statement examines the input host variables (if any) used in the definition of the cursor, determines the active set for the cursor, and leaves it in the open state, as described earlier. When SQL/DS executes an OPEN statement for a query-cursor, it positions the cursor *before* the first row of the active set. After the query-cursor is opened, SQL/DS does not re-examine its input variables until you close and re-open the cursor. No rows in the active set are actually fetched to the host program until a FETCH statement is executed.

If you are opening an insert-cursor and your program is blocking, this statement simply tells SQL/DS to prepare to block the rows that are to be inserted. With an insert-cursor, you can change the values of the input host variables between inserts. That is, you do not have to close and re-open the cursor in order to change the values to be inserted. Even if your program is not blocking, you should OPEN and CLOSE every insert-cursor.

Additional uses of the OPEN statement are described under "Dynamically Defined Statements" on page 147.

FETCH Statement

Format:

```
FETCH cursor-name INTO host-list
```

Example:

```
FETCH C1 INTO :NAME, :ADDR, :PHONE:PHONI
```

The FETCH statement can be executed only when the indicated cursor is in the open state. The position of the cursor is advanced to the next row of the active set, and the selected fields of this row are delivered into the output host variables

specified in the *host-list*. Output host variables in this list must be separated by commas, and must be immediately preceded by colons.

If the active set of the cursor is empty, or if all its rows have already been fetched, SQL/DS returns the “not found” SQL code (SQLCODE=100). To perform further operations via the cursor, you must close and re-open it.

Notice that the INTO clause on this statement is not optional; you must specify the output host variables in the FETCH statement, not in the cursor declaration. For example, the following is an invalid construction:

```
DECLARE QUERY1 CURSOR FOR
SELECT SUPPNO, PRICE*1.10
INTO :SUPP, :NEWPR
FROM QUOTATIONS
WHERE PARTNO = 221

OPEN QUERY1
FETCH QUERY1
```

Invalid. You should not use an INTO clause in a cursor declaration.

This is the correct way to specify the output host variables:

```
DECLARE QUERY1 CURSOR FOR
SELECT SUPPNO, PRICE*1.10
FROM QUOTATIONS
WHERE PARTNO = 221

OPEN QUERY1
FETCH QUERY1 INTO :S1, :P1
```

Correct. The values are returned in these host variables.

A cursor can move “forward” only when in its active set; SQL/DS provides no facilities for returning to rows that have already been fetched (other than closing the cursor and re-opening it).

It is possible for two or more rows in the active set to have exactly the same values. (For example, many rows of the QUOTATIONS table may have the same PARTNO, and you might define a cursor that selects only PARTNO from the table.) These duplicate values are not eliminated from the active set unless you specify DISTINCT in the SELECT clause of the DECLARE CURSOR statement.

You can use indicator variables in the INTO clause. In the above example, :PHONE:PHONI is a host variable (:PHONE) with an associated indicator variable (:PHONI). (See “Indicator Variables” on page 146 for a complete description.) Where nulls are applicable for a column, the value that SQL/DS returns in an indicator variable is coded as follows:

- 0 Denotes that the returned value is not null, and has been placed in the associated main variable.
- <0 Denotes that the returned value is null. The main variable should be ignored.

>0 Denotes that the returned value was truncated because the main variable was not of sufficient length.

In addition, if the truncated item was a DBCS or a character string, the indicator variable contains the length in characters before truncation. The SQLWARN1 warning flag in the SQLCA is set to 'W' whenever a returned character string is truncated. (See "Error Handling" on page 202 for a description of the SQLCA.)

Each main variable in the INTO clause may or may not have an associated indicator variable, at your option. If a null value is returned, and you haven't provided an indicator variable, a negative SQLCODE is returned to your program. If your data is truncated and there is no indicator variable, no error condition results.

Note that both the COMMIT WORK and ROLLBACK WORK statements automatically close all cursors.

When blocking is in effect, after a block of rows have been successfully retrieved from the data base, the variable SQLERRD(3) in the SQLCA indicates the number of rows retrieved. When blocking is not in effect, SQLERRD(3) is set to 1 after each successful FETCH. If the returned SQLCODE is non-zero, indicating unsuccessful completion of the statement, the content of SQLERRD(3) is unpredictable.

Additional uses of FETCH are discussed under "Dynamically Defined Statements" on page 147.

PUT Statement

Format:

```
PUT cursor-name
```

Example:

```
PUT C1
```

The PUT statement inserts one row of data into a table as defined by a cursor. This cursor must be defined in terms of an INSERT statement. The contents of input host variables (defined in the INSERT clause of the DECLARE CURSOR statement) are delivered to SQL/DS. These values are placed in the columns of the table that you specified.

For instance, the following statements insert a new row of data into the QUOTATIONS table. The values represented by the host variables :SUPP and :PART are placed in the SUPPNO and PARTNO columns of the new row. The other columns are assigned the null value.

```
DECLARE CC CURSOR FOR
INSERT INTO QUOTATIONS (SUPPNO, PARTNO)
VALUES (:SUPP, :PART)

OPEN CC
PUT CC
```

If you wish to, you can place constants in the VALUES clause of the DECLARE CURSOR statement, instead of host variables. However, this causes identical rows to be inserted for each PUT.

The PUT statement is used mostly for inserting multiple rows of data into a table in groups or blocks. However, the PUT statement also works with non-blocked inserts. Blocked inserts are specified with the BLOCK preprocessor parameter. If blocking is in effect, rows are not inserted into the data base until the block is full, or until a CLOSE statement is issued. For more information on blocking, see "To Block or Not to Block?" on page 254. For information on how to preprocess your program with the BLOCK option specified, see "Preprocessing the Program" on page 187.

The PUT statement can be executed only when the indicated cursor is in the OPEN state, otherwise SQL/DS returns a negative SQLCODE. Note that both the COMMIT WORK and ROLLBACK WORK statements automatically close all cursors.

After a block of rows have been successfully inserted using PUT statements, the variable SQLERRD(3) in the SQLCA indicates the number of rows inserted. When blocking is not in effect, SQLERRD(3) is set to 1 after each successful PUT. If the returned SQLCODE is non-zero, indicating unsuccessful completion of the statement, the content of SQLERRD(3) is unpredictable.

Additional uses of the PUT statement are discussed under "PUT Statement for Dynamically Defined Inserts" on page 181.

CLOSE Statement

Format:

```
CLOSE cursor-name
```

Example:

```
CLOSE C1
```

The indicated cursor leaves the open state, and its active set becomes undefined. No FETCH or PUT statement can be executed on the cursor, and no DELETE or UPDATE statement can refer to its current position, until the cursor is reopened by an OPEN statement. CLOSE permits SQL/DS to release the resources associated with maintaining an open cursor. CLOSE should be placed in your program so that it is executed as soon as the program is finished using a cursor.

If your program is blocking, closing an insert-cursor with an incomplete block will normally insert the remaining rows into the data base. Closing a query-cursor in this case will return the remaining rows in the incomplete block to the program.

It is recommended that you explicitly close all cursors before issuing a COMMIT WORK, especially when blocking.

Predicates

One of the most common operations in SQL is to search through a table, choosing certain rows for processing. A *search condition* is the criterion for choosing rows.

A search condition is a collection of one or more *predicates*. Each predicate specifies a test that SQL/DS applies to the rows of the table. You can connect predicates with the logical operators AND and OR. For example:

```
predicate1 AND predicate2 OR predicate3
```

The keyword NOT can be used to negate a predicate:

```
predicate1 AND NOT predicate2
```

The precedence rule among the keywords is as follows: first NOT is applied, followed by AND, followed by OR. You can use parentheses to override this precedence rule if necessary. For example, the search condition in Figure 5 contains three predicates; it could be used to find the rows of the QUOTATIONS table pertaining to supplier number 61 and part number 221 or 222:

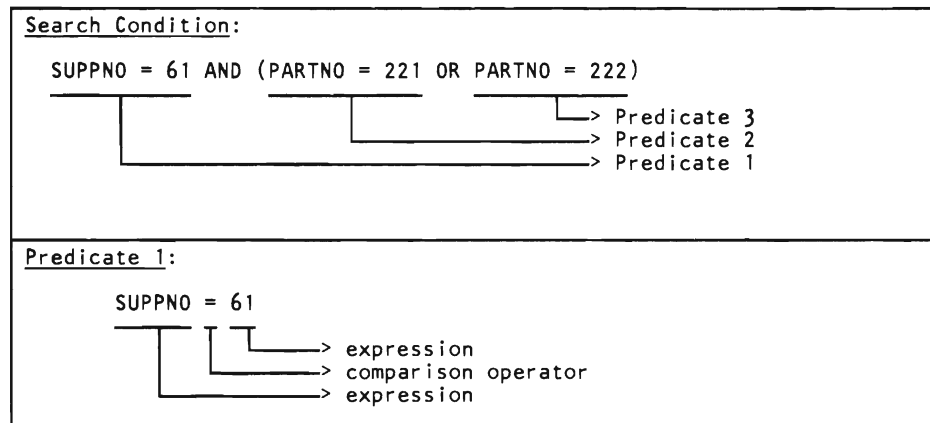


Figure 5. Breakdown of Search Conditions and Predicates

Figure 5 also shows that the format of a predicate is a comparison between two values or expressions. This format is represented as follows:

```
expression comparison-operator expression
```

A *comparison-operator* may be any of the following:


```

=      "equal to"
≠     "not equal to"
>     "greater than"
>=    "greater than or equal to"
<     "less than"
<=    "less than or equal to"

```

The above symbols are the only comparison operators that you can use in SQL/DS statements. For example, SQL/DS does *not* recognize “≠” even if supported in the host language. The correct representation of inequality is “- =.”

Host Variables and Constants

You know that a host variable is just a normal program variable by which SQL/DS interacts with the host program. You also know that they can be coded in the INTO clauses of SELECT statements or in the FETCH statements associated with cursors, in order to receive values selected from SQL/DS tables. But there are other places that you can use host variables. You can use them in WHERE clauses, for instance:

```

DECLARE C CURSOR FOR
SELECT SUPPNO
FROM QUOTATIONS
WHERE PRICE < :PRLIM AND PARTNO = :PART

```

You can also use them in other types of statements. For example, in a DELETE statement:

```

DELETE FROM QUOTATIONS
WHERE PARTNO = :XXX

```

(DELETE statements are discussed later in this chapter.)

When would you use the DECLARE CURSOR example above? Suppose that you had a list of part numbers and a corresponding list of the upper limits on prices that your company wants to pay for each part. Then you wanted to know which suppliers sell this part at a reasonable price. You could code the above statement in a loop which changes the part number and price limit on each pass. Then, for each pass through the loop, you would have a list of the supplier numbers that sell that part below your price limit.

Constants (also called *literals* or *literal constants*) can be numeric or character data. They are fixed values that can be coded into SQL statements. Like host variables, they are used in the SELECT and WHERE clauses of the SELECT and DECLARE CURSOR statements.

Numeric data can be integer, decimal or floating point data. Integer constants consist of a number with an optional sign, such as -56, 103, or +786. (If you do not include a sign, SQL/DS assumes that the number is positive.) Decimal data consists of a number with a decimal point, such as 78.9687, -.00132, 64570., or +1672.80. If you do not supply a decimal point, SQL/DS interprets the constant as an integer. A floating point number is an integer or a decimal constant followed by an exponent marked by the letter E. E must be followed by an exponent. E0 is

acceptable and evaluates to 1. All these are permissible floating point constants: -2E5, 2.2E-1, .2E6, +5E+2 or 4E0.

Character string constants are strings of letters or numbers, such as 'SMITH', '52', or 'k@r -5B'. They are considered varying-length character strings by SQL/DS. (Data types are discussed later in this chapter.) Character string constants must be put in single quotes when coded in an SQL statement. The following example shows a character string constant coded in a WHERE clause.

```
DECLARE C CURSOR FOR
SELECT *
FROM SUPPLIERS
WHERE NAME = 'DEFECTO PARTS'
```

Numeric constants can also be coded in the WHERE clause.

If you want to represent a single quote inside of a character string constant, use two single quote marks. SQL/DS interprets the constant:

```
'DON' 'T GO'
```

as:

```
DON'T GO
```

Constants, both character and numeric, can also be used in the SELECT clause. The effect of this is to set up a new column in the resulting display, which has the specified constant in each of its data fields. For example, the statement:

```
DECLARE C CURSOR FOR
SELECT NAME, 'WOW', 98.6
FROM SUPPLIERS
WHERE SUPPNO < 60
```

would have the following active set:

NAME	EXPRESSION 1	EXPRESSION 2
DEFECTO PARTS	WOW	98.6
VESUVIUS, INC.	WOW	98.6
ATLANTIS, CO.	WOW	98.6
TITANIC PARTS	WOW	98.6
EAGLE HARDWARE	WOW	98.6

For more information on constants and data types, see "Data Types" on page 75. Also see "Additional Types of Constants" on page 99 for a discussion of other types of constants that you can use within expressions.

Using Expressions as Search Conditions

In addition to column names, constants and host variables, any combination of these, connected by arithmetic operators, can also be used in SELECT and WHERE clauses. These are called *expressions*. An *expression* can be a column name, a constant, a host variable, or any arithmetic combination of these. Expressions allow you to do calculations on data as part of a query. The calculations are performed before SQL/DS returns the data to your program.

Figure 6 shows a simple expression:

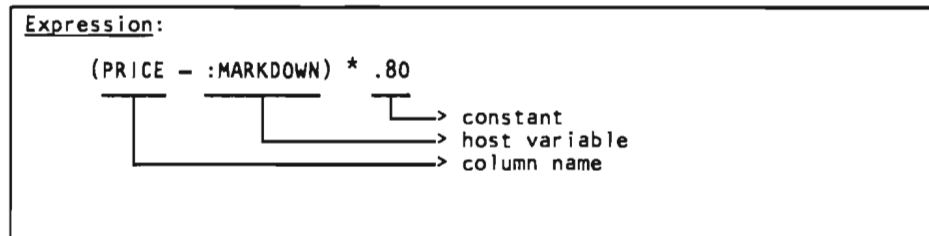


Figure 6. Breakdown of an Expression

There are four *arithmetic operators* that you can use:

- * multiplication
- / division
- + addition
- subtraction

Usually, SQL/DS reads the expression from right to left, first applying any negations, then any multiplication or division operations, then finally carrying out additions and subtractions. You can change this order or precedence by using parentheses. For instance, in the above example, if it were coded

```
PRICE - :MARKDOWN * .80
```

SQL/DS would take the value of the host variable MARKDOWN, multiply it by .80, and then subtract the result from the price. As the statement was originally coded, SQL/DS first subtracts MARKDOWN from PRICE and then multiplies the result by .80. The two results would probably end up being quite different.

You can use parentheses in an expression if you want to establish precedence among the operators. The default precedence rule is: negation is applied first, followed by multiplication and division, followed by addition and subtraction.

Host variables, as noted earlier, can be used in expressions either alone or in combination with other things. For instance

```
:QUANTITY
```

is a valid expression and so is

```
PRICE * :QUANTITY + 1.44
```

As mentioned earlier, you must precede the names of the host variables by a colon (:) to distinguish them from column names. That is, SQL/DS interprets

```
:PARTNO
```

as a host variable, but interprets

```
PARTNO
```

as a column name.

Numeric constants can stand alone or be used in arithmetic combination with other constants or host variables or column names. Thus, all three of the following are valid expressions:

```
200                -798.9768                PRICE * :QUANTITY + 1.44
```

On the other hand, *alphabetic constants* are only valid as expressions when they stand alone. They cannot be used in arithmetic combinations. Thus, these are two valid expressions using character constants:

```
'PLEASE DON'T EAT THE CANDY'                'BOLT'
```

whereas this is not a valid expression:

```
'FUDGE' * 'GUMDROP' + 'LEMON'
```

However, character constants can be used in comparisons in **WHERE** clauses. Thus,

```
WHERE NAME <= 'BOLT'
```

is a legitimate **WHERE** clause. In the above example, **NAME** represents a column name and 'BOLT' is a character constant. Remember that all character constants must be surrounded by single quotes.

If you attempt to combine, with arithmetic operators, two pieces of data that do not have compatible data types, SQL/DS will return an error code. All numeric data types are compatible with each other. SQL/DS performs data conversion on different types of data that are compatible. See "Data Conversion" on page 76 for more information on data conversion and compatibility.

Built-In Functions

You can also use the SQL/DS *Built-In functions* in the **SELECT** and **WHERE** clause(s) of **SELECT** statements. The built-in functions perform handy calculations for you, and present your program with the results, just like it was information retrieved from the data base.

SQL/DS has five built-in functions that you can use in expressions in select-lists:

```
AVG        MAX        MIN        SUM        COUNT
```

The argument of a built-in function may be a column name (optionally preceded by DISTINCT or ALL -- ALL is the default), or an expression. The argument follows the function and must be enclosed in parentheses.

DISTINCT indicates that duplicate values are to be eliminated before the function is applied. For example,

```
SELECT COUNT(DISTINCT PARTNO)
```

computes the number of different part numbers in the rows that satisfy the search condition. ALL indicates that duplicates are not to be eliminated.

The following are examples of SELECT statements using built-in functions:

```
SELECT AVG(PRICE)
INTO :MEAN
FROM QUOTATIONS
WHERE PARTNO = 222
```

```
SELECT MAX(PRICE) - MIN(PRICE)
INTO :DIFF
FROM QUOTATIONS
WHERE PARTNO = :PART
```

```
SELECT MAX(QONHAND+15)
INTO :MAXIMUM
FROM INVENTORY
```

```
SELECT MAX(PRICE * :DISCOUNT)
INTO :MAXDISC
FROM QUOTATIONS
```

```
SELECT COUNT(DISTINCT PARTNO)
INTO :NUM
FROM QUOTATIONS
```

```
SELECT MIN(PRICE), MAX(PRICE), MAX(PRICE) - MIN(PRICE)
INTO :H, :L, :SPREAD
FROM QUOTATIONS
WHERE PARTNO = 222
```

A special built-in function, COUNT(*), is also provided to count how many rows satisfy the search-condition. For example, the following query counts the rows of the QUOTATIONS table that apply to part number 222:

```
SELECT COUNT(*)
INTO :N
FROM QUOTATIONS WHERE PARTNO=222
```

You must follow these rules when using built-in functions:

1. In a select-list, built-in functions cannot be mixed with expressions that do not contain built-in functions. For example, SELECT PARTNO, AVG(PRICE) is an error. Exceptions to this rule are permitted in "grouping" type queries, which are described under "Grouping" on page 113.
2. In computing built-in functions such as AVG, SUM, MAX, and MIN, SQL/DS ignores null values. However, if SQL/DS encounters nulls in computing a

built-in function, it sets a warning flag (called SQLWARN2) in the SQLCA. The function COUNT(*) counts all rows that satisfy the search-condition, regardless of whether they contain null values.

3. If a built-in function is computed over an empty set (that is, if no rows satisfy the search condition), the following value is returned: COUNT returns zero; AVG, SUM, MAX, and MIN return the null value. (You should have an indicator variable to handle this condition.)
4. The built-in functions AVG and SUM can be applied to numeric columns (INTEGER, SMALLINT, DECIMAL or FLOAT type) only. If the data type of the operand is DECIMAL or FLOAT, the result of the function is the same data type as the operand column. If the data type of the operand is INTEGER or SMALLINT, the data type of the result is INTEGER. (In this case, if the true average is not an integer, the fractional part is truncated.) If the operand of SUM or AVG is DECIMAL with precision p and scale s , the result is precision 15. For SUM, the resulting scale is s . For AVG, the resulting scale is:

$$15 - p + s$$

For example, suppose you average a column having a data type of DECIMAL (5,2). The precision (p) is 5, and the scale (s) is 2. When SQL/DS averages the number, the resultant precision is 15 and the scale is $(15-5)+2$. Thus, the resultant scale is 12.

5. The built-in functions MAX and MIN may be applied to columns of any type. The result of these functions is always the same data type as the argument. If applied to a column of character-string type, dictionary ordering is used to find the MAX or MIN. For example:

```
'A' < 'B'  
'A' < 'ABLE'  
'Z' < '35'  
'A1' < 'B'
```

6. The built-in function COUNT can be used in only two ways:
 - a. COUNT(*) returns the number of rows that satisfy the WHERE clause.
 - b. COUNT(DISTINCT column-name) returns the number of *different* values of the given column in those rows that satisfy the WHERE clause. For example, COUNT(DISTINCT PARTNO) returns the number of different part numbers.

Note that you cannot apply COUNT to a column unless you also specify DISTINCT. For example, COUNT(PARTNO) results in an error. This is because the number of part numbers including duplicates is equal to the number of rows that satisfy the WHERE clause, which is correctly expressed by COUNT(*). The result of COUNT is always an integer. If the host variable into which the result of COUNT is placed does not have a data type of INTEGER, SQL/DS attempts to convert the result of COUNT into the data type of the host variable. (See "Data Conversion" on page 76.)

7. In a select-list, you can use the term **DISTINCT** only once. **DISTINCT** can be used to eliminate duplicates from the query result as a whole (**SELECT DISTINCT PARTNO,PRICE**). Alternatively, it can be used to eliminate duplicates from the argument of a function (**SELECT COUNT(DISTINCT PARTNO)**). However, you cannot mix these usages.
8. If you use **DISTINCT** inside the argument of a function, the argument must be a simple column name, not an expression. (For example, **COUNT(DISTINCT QONHAND/2)** is not permitted.) Also, a function with **DISTINCT** in its argument must stand alone, and cannot be used inside an expression such as **COUNT(DISTINCT PARTNO)+10**.
9. Although **COUNT(*)** includes the number of rows whose values are null, **COUNT(DISTINCT...)**, **AVG**, **MAX**, **MIN**, and **SUM** ignore null values.

Putting a New Row into a Table: INSERT

Format 1 INSERT:

```
INSERT INTO [creator.]table-name [(list-of-column-names)]
VALUES (list-of-data-items)
```

Examples:

```
INSERT INTO JONES.INVENTORY (PARTNO,DESCRIPTION,QONHAND)
VALUES (251,'GEAR',:QOH:IND1)
INSERT INTO QUOTATIONS VALUES (:A,:B,:C:CI,:D:DI,:E:EI)
INSERT INTO QUOTATIONS VALUES (68,209,18.00,14,0)
INSERT INTO WEATHER (DATE, LOCATION, TEMPERATURE)
VALUES ('JANUARY 13, 1981','ENDICOTT',-15)
```

Authorization:

You can insert data into any table you create. You can insert data into another user's table if you are given the **INSERT** privilege on that table, or if you have **DBA** authority.

Format 1 of the **INSERT** statement adds a single row of data into an existing table. The statement consists of two clauses: the **INSERT** clause and the **VALUES** clause. In the **INSERT** clause, specify the name of a table (*table-name*) and (optionally) a list of column names (*list-of-column-names*) that the data is to be inserted into. In the **VALUES** clause, place the values that you want added to the table in the *list-of-data-items*. Separate each item with a comma.

SQL/DS forms new rows by placing the various data-items into the specified columns in the order named:

```
INSERT INTO SUPPLIERS (SUPPNO, NAME)
VALUES (68, 'EAGLE HARDWARE')
```

In the example above, SQL/DS places 68 in SUPPNO and 'EAGLE HARDWARE' in NAME. You do not have to list the column names in the same sequence that they were named when the table was created. For example, this statement is equivalent to the previous one:

```
INSERT INTO SUPPLIERS (NAME, SUPPNO)
VALUES ('EAGLE HARDWARE', 68)
```

Omitting the list of column names is the same as naming all the columns in the order that they were named when the table was created. If you **do** include the list of column names, all columns of the given table that you do not name receive the null value. You can also insert null values into a table by using the NULL keyword:

```
INSERT INTO INVENTORY
VALUES (291, 'LEVER', NULL)
```

In the above example, omission of the column-list denotes that all columns participate; but the last column, QONHAND, receives a null value because of the NULL keyword in the list of data-items. If you attempt to insert nulls into a column that does not permit nulls, SQL/DS returns an error code in the SQLCA.

For the list-of-data-items, you can use constant (literal) values such as 'JOHN DOE' or -750. You can also use host variables such as :X or :PART.

The data types of the values to be inserted (source data type) do not necessarily have to match the data types defined for the columns (target data type). However, the data types must be compatible, that is, character to character, numeric to numeric, or DBCS to DBCS. SQL/DS automatically does data conversion on compatible data types. (See "Data Conversion" on page 76 for more information.)

SQL/DS uses no logical ordering on the rows of a table. Therefore, you cannot specify a "position" in the table for the new row. SQL/DS just associates the new row with the rest of the table. When a SELECT statement is next issued on that table, SQL/DS determines the row's position by checking available indexes on the table and by following sort instructions listed in the SELECT statement. (Indexes and SELECT sort instructions are discussed in Chapter 2.)

Deleting Data from a Table: DELETE

Format 1 DELETE:

```
DELETE FROM [creator.]table-name  
    [WHERE search-condition]
```

Examples:

```
DELETE FROM QUOTATIONS WHERE SUPPNO = 53  
DELETE FROM QUOTATIONS WHERE DELIVERY_TIME IS NULL  
DELETE FROM QUOTATIONS WHERE PARTNO = :X AND PRICE > :Y  
DELETE FROM SCOTT.INVENTORY WHERE DESCRIPTION = 'PISTON'
```

Authorization:

You can delete rows from any table you create. You can delete rows from another user's table if you are given the DELETE privilege on that table, or if you have DBA authority.

You can delete one or more rows of data from a table by using the DELETE statement. SQL/DS deletes all rows of the named table that satisfy the search conditions that you specify. For instance, the following example deletes all the rows in the QUOTATIONS table that have a supplier number of 53.

```
DELETE FROM QUOTATIONS  
WHERE SUPPNO = 53
```

The DELETE statement has two clauses: the DELETE clause and the WHERE clause. The DELETE clause consists of the keywords DELETE FROM followed by the name of the table that you want the rows deleted from. The WHERE clause is made up of the keyword WHERE followed by a search condition. This WHERE clause is just like the WHERE clause in the SELECT statement. The search-condition simply describes the rows that you want SQL/DS to search for and, in this case, delete. For more information on the WHERE clause and search conditions, see "WHERE Clause: Searching on Conditions" on page 19.

If you omit the WHERE clause, SQL/DS deletes **all the rows** from the indicated table. For instance, the statement

```
DELETE FROM SUPPLIERS
```

would delete all the rows from the SUPPLIERS table. The table would still "exist," but would be empty until you issue a DROP TABLE statement. When this happens, SQL/DS sets a warning indicator in the SQLCA (SQLWARN4). You can check this warning indicator to detect unintentional deletions and, if necessary, you can undo these deletions before they are permanently committed to the data base. (See "ROLLBACK WORK" on page 233 and "Error Handling" on page 202 for more information.)

If no rows satisfy the search condition, SQL/DS returns a message (SQLCODE=100) in the communications area that you declared in your application prolog. It does not delete any rows.

If SQL/DS detects an error in your DELETE statement after some rows have already been deleted, SQL/DS stops processing the statement and returns an error code in the SQLCA.

After successful completion of a DELETE statement, the variable SQLERRD(3) in the return code structure indicates the number of rows that were deleted. If the returned SQLCODE is non-zero, indicating unsuccessful completion of the statement, the content of SQLERRD(3) is unpredictable.

Changing Data in a Table: UPDATE

Format 1 UPDATE:

```
UPDATE [creator.]table-name
SET column-name-1 = expression-1
[, column-name-2 = expression-2] ...
[ WHERE search-condition ]
```

Example:

```
UPDATE EMPLOYEES
SET SALARY = 65000.00,
    POSITION = 'RETIRED'
WHERE NAME = 'J. B. ROBINSON'
```

```
UPDATE SUPPLIERS
SET NAME = :NAM:INAM,
    ADDRESS = :ADDR:IADDR
WHERE SUPPNO = :SNO
```

Authorization:

You can update tables you create. You can update columns in other user's tables if you are given the UPDATE privilege on the columns, or if you have DBA authority.

A Format 1 UPDATE statement changes the values of one or more fields in one or more rows of a table. All rows that satisfy the search condition are updated. For example, the following statement adds the content of variable X to the QONORDER field of the row for part number 231 in the QUOTATIONS table:

```
UPDATE QUOTATIONS
SET QONORDER = QONORDER + :X
WHERE PARTNO = 231
```

The UPDATE statement consists of three clauses: the UPDATE clause, the SET clause and the WHERE clause.

The UPDATE clause contains the name of the table that you wish to update. If this table belongs to another user, you must concatenate the owner's userid to the table name.

The SET clause specifies the changes you wish to make to particular columns of the chosen row(s). One or more fields in each row have their values replaced by the value of an expression. An expression can be a constant, a host variable, a column name, or any combination of three, joined by the arithmetic operators +, -, *, and /.

The following example sets the PRICE field to 2500.00/QONORDER and then sets the QONORDER field to zero of the row for part number 525 in the QUOTATIONS table:

```
UPDATE QUOTATIONS
SET PRICE = 2500.00 / QONORDER, QONORDER = 0
WHERE PARTNO = 525
```

The above example also illustrates the following rule: SQL/DS computes all update values *before* any updates become effective. Thus, SQL/DS computes the new value of PRICE before setting QONORDER to zero, regardless of the order in which you list the individual updates in the SET clause.

As with the INSERT statement, if data types in the SET clause are compatible but not identical, SQL/DS applies data conversion. Data conversion rules are discussed under "Data Conversion" on page 76.

The WHERE clause is just like the WHERE clause in a SELECT statement -- it specifies which rows are to be updated. If you omit the search condition, SQL/DS updates **all the rows** in the named table. However, when this happens, SQL/DS sets a warning indicator in SQLWARN4 of the SQLCA so that you can detect unintentional updates and, if necessary, undo these changes before they are permanently committed to the data base. (See "ROLLBACK WORK" on page 234 and "Error Handling" on page 202 for more information.)

If no rows satisfy the search condition, the "not found" code (SQLCODE=100) is returned in the SQLCA. No rows are updated.

If SQL/DS detects an error in your UPDATE statement after some rows have been updated (for example, an attempt to update a NOT NULL field to NULL), SQL/DS stops processing the statement and returns an error code in the SQLCA.

You can set the contents of a field to the null value by writing column-name = NULL in the SET clause of an UPDATE statement. You can also set a field's contents to null by using an indicator variable. The following example updates the INVENTORY table and sets the QONHAND field to null for a certain part:

```
UPDATE INVENTORY
SET   DESCRIPTION = 'INACTIVE',
      QONHAND = NULL
WHERE PARTNO = 801
```

You can improve the performance of UPDATE statements if you do not update the same column on which you are searching. Suppose there is a table EMP that contains a column NAME and a column NUMBER. Each name has a unique person-number. If you want to update the name field, you should code the update as:

```
UPDATE EMP SET NAME='new name'      <---- Fast
           WHERE NUMBER=value
```

The above statement is much faster than this one:

```
UPDATE EMP SET NAME='new name'      <---- Slow
           WHERE NAME='old name'
```

After successful completion of an UPDATE statement, the variable SQLERRD(3) in the SQLCA indicates the number of updated rows. If the returned SQLCODE is non-zero, indicating unsuccessful completion of the statement, the content of SQLERRD(3) is unpredictable.

Preprocessing and Running the Program

This section gives you an introduction to the steps it takes to prepare and run your program. It tells you what it means to preprocess, compile, load and run an SQL/DS application program.

Note: Readers should already know how to compile, load and execute programs in their host languages. This section, and the corresponding “Preprocessing and Running the Program” in Chapter 2, only cover the peculiarities of compiling, loading and executing SQL/DS application programs.

Contents

Section Quiz	42
Answers to the Section Quiz	43
Introduction	44
Preprocessing the Program	44
Compiling the Program	45
Link-Editing and Loading the Program	45
Running the Program	45

Section Quiz

The questions in this quiz cover the high points of this section. If you can answer most or all of these questions, you probably do not need to read this section. If you decide to skip ahead, proceed to "Testing and Debugging Concerns" on page 47. If you have trouble answering these questions, you should read the whole section.

1. What are the four steps necessary to prepare and run your program?
2. What are the two things that preprocessing your program does?
3. TRUE or FALSE: Compiling an SQL/DS application program is no different than compiling an ordinary program in your host language?
4. What is an access module?
5. What is the difference between multiple user mode and single user mode?

Answers to the Section Quiz

1. 1. Preprocessing the SQL code, 2. Compiling the program, 3. Link-editing and loading the program, 4. Running the program.
2. It changes the SQL source code so that it can be processed during host language compiling and converts the SQL statements into an “access module” that is stored in the SQL/DS data base.
3. TRUE. However, there are some minor exceptions. See “Preprocessing and Running the Program” on page 183 for an account of these exceptions.
4. An access module is a machine code version of the SQL requests made by your program, stored in the SQL/DS data base.
5. Multiple user mode allows one or more users or programs to access the same data base at the same time. Single user mode only allows one user or program to access the data base at a time.

Introduction

Once your program is coded, you must get it ready to be run. In SQL, this involves a series of steps. The number of steps varies depending on the host language of the program and the environment in which the program is running. There are, however, four steps that are common in each case. In order to run your SQL application program you must:

1. Preprocess the SQL code.
2. Compile the program.
3. Link-edit and load the program.
4. Run the program.

Now let's look at each of these steps in a little more detail.

Preprocessing the Program

Preprocessing your SQL code does two things:

- It changes the SQL source code so that it can be processed during host language compiling.
- It converts the SQL statements into an "access module" that is stored in the SQL/DS data base.

The preprocessor replaces all the SQL statements in the program with host language code that invokes the new access module. The new version of the program also contains the SQL statements in comment form. The access module contains machine code to carry out the SQL requests made by the program. SQL/DS chooses the best access path to the data for each SQL command in the program, basing its choice on available indexes and data statistics that SQL/DS keeps track of.

When the program is run, the new code calls the module to handle each SQL command. It also links the program to SQL/DS and translates messages and commands between the two.

If the preprocessor encounters a severe error in an SQL statement, only syntactical checking is performed on subsequent SQL statements. It also puts statements in the preprocessed program which will cause a subsequent compile to fail.

Compiling the Program

Once you have successfully preprocessed your program, you can compile it using your normal host language compiler. By preprocessing the program, you have already done all the translating that the program needed. Just use the new code that you got after you preprocessed. Compile this code just like you would any other program, using the usual compilers.

You should know how to compile a program in your host language already. This book does not cover the specifics of compiling your host-language code. However, there are a couple of special rules for SQL programs, depending on the host language, that you must follow. These rules are discussed in “Compiling the Program” on page 196.

Link-Editing and Loading the Program

After compilation, programs must be link-edited and loaded before they can be run. To allow your program to communicate with SQL/DS, you must link-edit your program with one or more SQL/DS TEXT files. One of these TEXT files is called the *resource manager stub*. Every SQL/DS application program must be link-edited with this stub. FORTRAN and COBOL programs need to link-edit with an additional TEXT file. Also, depending on the nature of your program, you may have to link-edit with others.

One way to link-edit these TEXT file(s) successfully to your program is to INCLUDE the TEXT filename(s) after your program name in the CMS LOAD command. Then, when you load your program, the CMS linkage editor automatically links your program to the TEXT files that you specified and resolves virtual storage addresses between files.

See Chapter 2 for more information on link-editing and loading.

Running the Program

Once you have loaded your program, it is ready to be run. You can run your program in either single user mode or multiple user mode. In single user mode, SQL/DS, its preprocessors, and your application programs all run in a single VM/SP virtual machine. This is also sometimes referred to as single virtual machine mode. Multiple user mode allows one or more users or programs to access the same data base at the same time. This is sometimes referred to as multiple virtual machine mode.

How you execute your SQL/DS program depends on the mode in which SQL/DS is running. You can find the details of this under “Running your Program” on page 198.

The access module that the preprocessors stored in the data base actually carries out the SQL request. When SQL/DS loads the access module, it checks to see that

the access module is still valid. An access module may not be valid if it lost some dependency. For example, some index that the access module uses may have been dropped. SQL/DS has an internal change management facility that keeps track of which access modules are valid and which are not valid.

If the access module is valid, SQL/DS begins running the program. If the access module is not valid, SQL/DS tries to recreate it. The original SQL statements are stored with the access module when you preprocess the program. SQL/DS uses these SQL statements to try to automatically preprocess the program again. It does this dynamically; that is, as it is running. If this "re-preprocessing" works, a new access module is created and stored in the data base. SQL/DS then continues execution of the program. If the re-preprocessing does not work, SQL/DS returns an error code to the program in the SQLCA, and the program stops running.

The re-preprocessing, if it succeeds, has no negative effect on your program except for a slight delay in processing your first SQL statement.

All the details of getting your program ready to be run are in "Preprocessing and Running the Program" on page 183.

Testing and Debugging Concerns

This section gives you an introduction to two methods of testing and debugging your SQL/DS application. The first method is testing SQL commands online, before you actually code them into the program. The second method makes use of the SQL Communications Area (SQLCA), which is the automatic SQL/DS error handling facility.

Contents

Section Quiz	48
Answers to the Section Quiz	49
Introduction	50
Using ISQL to Test SQL Statements Before Coding	50
Introduction to the SQL Communications Area (SQLCA)	51

Section Quiz

1. What online SQL/DS facility allows you to test commands to see if they are valid, before you code them in your application program?
2. What is a logical unit of work?
3. What are the two steps you must do to tell SQL/DS what action to take when it comes across an SQL error?
4. What does an SQLCODE of 0 mean? What does a negative SQLCODE mean? A positive SQLCODE?



Answers to the Section Quiz

1. ISQL
2. A group of SQL statements, possibly with intervening host language code, that are treated as a single unit or entity.
3. You must declare an SQL Communications Area and code an SQL **WHenever** statement.
4. An **SQLCODE** of 0 means that an SQL statement has executed successfully. **SQL/DS** indicates error conditions by returning a negative **SQLCODE**. A positive **SQLCODE** indicates normal conditions experienced while executing the statement (such as end-of-file).

Introduction

Of course, even the best programmers make mistakes in coding. Unfortunately, you have to correct these errors before the program will run correctly. Thus, you must have methods for checking your code, to make sure it is valid.

In SQL, there are many ways to test your SQL statements and debug them. Some of these methods are done automatically by SQL. For example, during preprocessing, if the preprocessor comes across an SQL error, it inserts statements in the new source code that show this. Then when you try to assemble or compile that code, these error statements halt the compilation and tell you there was an error.

Using ISQL to Test SQL Statements Before Coding

There are other methods of error testing that you can do on your own. One such method is using the Interactive Structured Query Language (ISQL) facility to test your SQL statements before you code them into the program. This method lets you see the results of a command on the screen as you work interactively with the SQL/DS data base. This way you can't disrupt your program during testing. All your testing is done on the screen. If the command works, code it in your program; if not, debug it right on your terminal until it does work.

In ISQL, all commands are entered at the terminal in basic form. None of the host language delimiters are added. Also, you cannot enter any cursor commands of any kind (DECLARE, OPEN, FETCH, PUT, or CLOSE). In addition, "programming-only" statements such as declarative statements and dynamically-defined statements cannot be entered in ISQL.

The statements that you would most often want to test through ISQL are:

- SELECT
- INSERT
- UPDATE
- DELETE

On the other hand, you may wish to handle most of your data definition, authorization, and data control tasks through ISQL. It is often easier to define the tables and store the data for your program *first*, through ISQL, and then to operate on that data, through programs, *later*. Also, in most cases, it is recommended that you grant and revoke authorizations on tables and programs through ISQL, and *not* through application programs.

For a tutorial on how to use ISQL, refer to *SQL/Data System Terminal User's Guide for VM/SP*, SH24-5045. In addition, *SQL/Data System Terminal User's Reference for VM/SP*, SH24-5067, contains reference information on all the commands that you can issue in ISQL.

You can also use SQL in the control data set of the Data Base Services (DBS) utility. This is discussed in the *SQL/Data System Data Base Services Utility for*

VM/SP, SH24-5069 manual. In addition to its data loading and unloading capabilities, the DBS utility processes SQL statements in a manner similar to ISQL, although not interactively. You can use either ISQL or the DBS utility to create test tables for your programs.

Introduction to the SQL Communications Area (SQLCA)

Every SQL application program must provide for error handling by declaring an SQL Communications Area. This area receives messages that SQL/DS sends to the program. By testing certain fields of this area, you can test for certain conditions during the program's execution.

Error handling is important in SQL/DS because it helps protect the integrity of the data base when a program fails. For example, consider the two-step operation needed to transfer \$500 from one account to another in a bank:

1. Subtract \$500 from account A
2. Add \$500 to account B.

If the system or your program fails after the first statement is executed, some customer has just "lost" \$500. This type of incomplete update is said to leave the data base in an *inconsistent state*.

You can avoid an inconsistent state by using a logical unit of work. A *logical unit of work* is a group of related SQL statements, possibly with intervening host language code, that you wish to treat as a unit. The two steps in the previous example would make up a single logical unit of work.

Logical units of work prevent inconsistent states from system or SQL statement errors. For system errors, SQL/DS automatically restores all changes made during the logical unit of work in which it encountered the error. This is called a roll back. For SQL errors, you must tell SQL/DS what action to take when it comes across an SQL error. This involves two steps:

1. Declaring an SQL Communications Area
2. Coding an SQL WHENEVER statement.

To declare the SQL Communications Area (SQLCA), code this statement in your program:

```
INCLUDE SQLCA
```

When you preprocess your program, SQL/DS inserts host language variable declarations in place of the INCLUDE SQLCA statement. This group of variables is how SQL communicates with your program. SQL/DS uses the variables for warning flags, error codes and diagnostic information. All the variables are discussed under "Error Handling" in Chapter 5. The only variable you need be concerned with now is SQLCODE.

SQL/DS returns a result code in SQLCODE after executing each SQL statement. SQLCODE, return code, and result code are all terms that mean the same thing: the integer value that summarizes how your SQL statement executed. When a statement executes successfully, SQLCODE is set to 0. SQL/DS indicates error conditions by returning a negative SQLCODE. A positive SQLCODE indicates normal conditions experienced while executing the statement (such as end-of-file).

The WHENEVER statement below tells SQL/DS what to do when it encounters an SQL error (that is, a negative SQLCODE):

```
WHENEVER SQLERROR GO TO ERRCHK
```

That is, whenever an SQL error (SQLERROR) occurs, program control is transferred to a subroutine named ERRCHK. This subroutine should include logic to analyze the error indicators in the SQLCA. Depending on how ERRCHK is defined, action may be taken to execute the next sequential program instruction, to carry out some special functions, or, as in most cases, to roll back the current logical unit of work and end the program.

You can have any number of logical units of work in a program. For the simplest case (which is being discussed here) the whole program is a single logical unit of work. Either the program runs successfully and the changes are made to the data base, or it doesn't and no changes are made.

SQL/DS begins a logical unit of work implicitly. That is, you don't have to code a statement to start a logical unit of work. SQL/DS starts one when it encounters your first executable SQL statement.

You must tell SQL/DS when to end the logical unit of work. "Application Epilog" on page 93 explains how to do this. There are times when SQL implicitly ends a logical unit of work. When this occurs, the SQLWARN0 and SQLWARN6 indicators are set to 'W'.

Putting the Program into Production

Putting your program into production involves creating and controlling the data that your program works with. This also involves, optionally, granting privileges to other users to work with your data or run your program, and revoking these privileges when they are no longer necessary or useful. You must also keep track of which pieces of data, including programs, you own and which users have authority to access those pieces of data. All of these topics are introduced in this section.

Most data administration can be done using the Interactive SQL facility (ISQL). For more information on this facility, see *SQL/Data System Terminal User's Guide for VM/SP*.

Contents

Section Quiz	55
Answers to the Section Quiz	56
Authorization	57
Privileges on Tables and Views	57
Privileges on Programs	58
Special Privileges	60
Granting Privileges to Other Users	62
Revoking Privileges from Other Users	66
Data Control	70
How the Data Base Is Structured	70
Logical Units of Work	72
Dropping a Program	73
Data Definition	74
General Rules for Naming Data Objects	74
Data Types	75
Data Conversion	76
Qualifying Table Names	78
SQL/DS Catalogs	78
Catalogs that Record Privileges	79
SYSUSERAUTH	79
SYSUSERLIST	79
SYSPROGAUTH	80
SYSTABAUTH	80
SYSCOLAUTH	80
Catalogs that Record the Contents of the Data Base	80
SYSDBSPACES	80
SYSCATALOG	81

SYSACCESS	81
SYSVIEWS	81
SYSCOLUMNS	81
Catalogs that Record Indexes and Synonyms	81
SYSINDEXES	81
SYSSYNONYMS	81
Miscellaneous Catalogs	81
SYSUSAGE	81
SYSDROP	82
SYSCHARSETS	82
SYSOPTIONS	82

Section Quiz

If you can answer most of the following questions, then you probably do not have to read this section. If you choose to skip ahead, proceed to Chapter 2. If you have trouble answering the questions in this quiz, proceed to “Authorization” on the next page.

1. Write an SQL statement that would grant the RUN privilege on a program called LISTING to user KIM. Also give KIM the privilege to grant RUN authority on this program to other users.
2. Write an SQL statement that would take away from user JULIE the privilege to insert rows into your ACCOUNTS table.
3. Write an SQL statement that would delete a program called BANKING from your DBSPACE.
4. What is the maximum length (in characters) of a table name? What is the maximum length (in characters) of a program name?
5. With which characters must an SQL identifier begin?
6. Which SQL/DS catalog contains information on the privileges of users to run programs? Which catalog would you look at to get a complete list of the programs that you own?

Answers to the Section Quiz

1. GRANT RUN ON LISTING TO KIM WITH GRANT OPTION
2. REVOKE INSERT ON ACCOUNTS FROM JULIE
3. DROP PROGRAM BANKING
4. 18; 8
5. An uppercase letter (A-Z), \$, #, or @. (If an identifier is enclosed in double quotes, it may also begin with a number.)
6. SYSPROGAUTH; SYSACCESS

Authorization

SQL/DS keeps track of which *privileges* each user has, and makes sure that each user performs only authorized operations on the data base.

SQL/DS makes it easy for authorized users to create and drop tables, and to compile and run programs that operate on these tables. An individual who creates a table or compiles a program can selectively share the use of that table or program with other users.

When SQL/DS is installed, at least one person is given *Data Base Administrator* (DBA) authority. A user having DBA authority has control of SQL/DS resources and of all privileges to use SQL/DS. One of these privileges is the ability to pass on DBA authority to other users; thus, there may be many users with DBA authority in your installation.

Before you can perform any data base operations, you must be authorized to use SQL/DS. This special privilege is called CONNECT authority. Normally, you get CONNECT authority by having a DBA grant it to you, but a DBA can also grant CONNECT to "ALLUSERS". This makes it possible for anyone to be implicitly connected, but has some significant limitations. Implicit connect is discussed under "VM/SP Connect Considerations" on page 186.

Other privileges you need vary depending on what SQL/DS operations you want to perform. There are three categories of privileges: privileges on tables and views, privileges on programs, and special privileges. The following sections discuss each category.

Privileges on Tables and Views

You can have any or all of the following privileges on specific tables and views:

- SELECT** Privilege to retrieve data
- INSERT** Privilege to insert new rows
- DELETE** Privilege to delete rows
- UPDATE** Privilege to change field values
- ALTER** Privilege to add new columns to a table (does not apply to views or DBSPACES)
- INDEX** Privilege to create new indexes on a table (does not apply to views).

When you create a new table, you are automatically given full privileges on the table. SQL/DS also gives you the GRANT option on each privilege. You can grant these individual privileges, or any combination of them, to other users by a GRANT statement (described later). When you grant a privilege to another user, you may include the GRANT option. If you do, the user will be able to grant the privilege to others. Once granted, you may revoke a privilege by issuing a REVOKE statement (also described later). If you revoke a privilege from User A,

you automatically revoke it from all users to whom User A granted it. If the other users have another independent source for the same privilege, they are unaffected by the revocation.

For each privilege, you can also hold the GRANT option. Having the GRANT option means that you can grant the privilege to other users and exercise it yourself.

You may exercise any privilege that you hold on a table directly through ISQL (via the terminal) and the DBS utility as well as application programs.

Except for ALTER and INDEX, the same kinds of privileges that apply to tables also apply to views. As with tables, the user who defined the view gets certain privileges that can be selectively shared with other users.

Users' privileges on tables and views are listed in the SQL/DS catalogs SYSTABAUTH and SYSCOLAUTH. All SQL/DS catalogs are described in the *SQL/Data System Planning and Administration for VM/SP* manual. You can find out which privileges you hold, and which privileges you have granted to other users, by making suitable queries on these catalog tables.

Privileges on Programs

All SQL/DS application programs must be preprocessed. The preprocessor creates an *access module* and stores it in the data base. Access modules contain machine code to carry out SQL requests made by the application program; for SQL/DS, it is the essence of the application program.

When you successfully preprocess a program, you receive the RUN privilege on your program. This means that you may at any time run your program, which in turn loads and executes the appropriate access module. SQL/DS considers the *creator* (or author) of a program to be the value specified in the USERID preprocessor parameter. This creator is considered to be the connected user at the time that the program is preprocessed. The USERID preprocessor parameter establishes the userid that is to be checked for authorization to do the SQL/DS functions that are found in the program by the preprocessor.

Normally, authorization to perform SQL/DS functions is checked and found to be valid at preprocessing time. Even if some authorization is not found at preprocessing time, the author is still given RUN authority for the program. The missing authority is automatically rechecked at run time. If the required authority is still not in place at run time, execution is not permitted. A program containing an unauthorized statement runs successfully as long as it does not attempt to execute the unauthorized statement.

If the creator of a program receives authorization required by the program between the time of its being preprocessed and its being executed, SQL/DS commands affected by the newly granted authorization will execute more slowly than they would if the authorization had been available at the time of preprocessing. This slower processing can also be avoided by repeating the preprocessing of the program after the authorization is granted.

Generally speaking, if the program contains no statements that require DBA authority, and if the *creator* of the program:

1. Has all the privileges required for all the SQL statements in the program,
2. Has the GRANT option on all these privileges,

then the creator receives the RUN privilege on the program with the GRANT option. (See “Putting the Program into Production” on page 211 in Chapter 2 for additional information.) This enables the creator to grant the RUN privilege on that program to other users, thus providing authorization control on an application basis.

For example, suppose user Smith has the privilege to update employee salaries. Smith wants to authorize Jones to update salaries in a particular way, with certain record-keeping and validity checking. Smith can write a program that updates salaries subject to the desired constraints, and grant the RUN privilege on the program to Jones. Now Jones can update salaries by running the program, but does not have an unconstrained update privilege on salaries. What is really granted to Jones is the ability to invoke the access module for the indicated program. Note that SQL/DS protects only the access module (which implements the SQL statements in the program), not the logic of the program itself.

It is important to understand this distinction between the creator (author) of a program and the user who runs the program. The runner is the user who executes the program and therefore invokes the access modules associated with it. The runner is identified to SQL/DS through the CONNECT statement in the program or through the equivalent implicit connect function in the VM/SP environment. Generally, the authorization of the creator determines whether a particular SQL/DS statement may be executed. The only exception to this is in dynamically defined statements. Because the statements are processed at execution time, SQL/DS bases the authorization checking for these statements on the runner’s userid (not the creator’s).

In some cases you receive the RUN privilege with the GRANT option only if you explicitly have all the needed authority. Explicit authority means that there is an explicit entry in the SQL/DS catalogs recording the authority for the object. Suppose, for example, that you have DBA authority and preprocess a program that creates an INDEX for another user. If the user does not grant you INDEX authority, you receive only the RUN privilege. You do not receive the GRANT option because you do not explicitly have all the needed privileges. You will, however, be able to successfully execute your program because of your DBA authority.

In other cases the creator of a program may receive RUN privilege with the GRANT option even if that creator does not have the required privilege when the program is preprocessed. In this case the creator of the program, or anyone granted the RUN privilege, can run the program, once the privileges have been obtained from the creator. Refer to “Putting the Program into Production” on page 211 for decision tables that show this determination for each SQL statement type.

If you write a program that operates on some table that does not exist at preprocessing time (for example, a program to load data into a table that has not yet been created), it will not prevent you from receiving RUN privilege on the program. However, when the program is run, the table in question must exist, and you must have the necessary privileges to operate on it.

All the RUN privileges held by users on programs are listed in the SQL/DS catalog SYSPROGAUTH. By querying SYSPROGAUTH, you can find out which programs you are entitled to run, and which programs you have granted to other users. If you have DBA authority, you can run any program regardless of what is indicated in SYSPROGAUTH. All SQL/DS catalogs are described in the *SQL/Data System Planning and Administration for VM/SP* manual.

Note: Some SQL statements do not require an access module to be created by the preprocessor; therefore, RUN authority may not apply (if the access module is not created). The following SQL statements do not involve an access module when they are preprocessed:

CONNECT	Extended DESCRIBE
Extended PREPARE	CREATE PROGRAM
Extended DECLARE CURSOR	DROP STATEMENT
Extended OPEN/CLOSE/FETCH/PUT	WHENEVER

If you need more details about how SQL/DS authorizes programs, see “Putting the Program into Production” on page 211. That appendix contains decision tables for SQL statements that affect program authorization.

Special Privileges

In addition to privileges on tables and programs, SQL/DS recognizes some special privileges: CONNECT, SCHEDULE, RESOURCE, and DBA authority. As discussed above, CONNECT authority is the privilege of being recognized by SQL/DS for purposes of using the system. It means that there is a userid recorded in the SQL/DS catalogs for purposes of recognition. There may also be a password recorded with the userid. If there is a password with the userid, an explicit CONNECT statement is permitted. Without a password, only implicit connects are possible. (See “VM/SP Connect Considerations” on page 186 for more information on implicit connects.) Only a user with DBA authority can grant CONNECT authority to SQL/DS users.

SCHEDULE authority is the privilege to connect users without specifying a password. Although it is possible to grant and revoke SCHEDULE authority, SQL/DS ensures that only resource managers can use it. The SQL/DS online resource manager uses SCHEDULE authority when it connects a user to SQL/DS implicitly. More information about SCHEDULE authority is in the *SQL/Data System Planning and Administration for VM/SP* manual.

Resource authority permits you to create tables in PUBLIC DBSPACES and acquire PRIVATE DBSPACES. Resource authority is not required to create tables in your own PRIVATE DBSPACE.

A DBA can allow table creation by:

1. Granting RESOURCE authority to a user, or
2. Creating a PRIVATE DBSPACE for a user.

The latter offers more limited capability.

Only a DBA can create tables in PRIVATE DBSPACES owned by another user or acquire PUBLIC DBSPACES.

Holders of DBA authority automatically hold RESOURCE and CONNECT authority as illustrated in Figure 7.

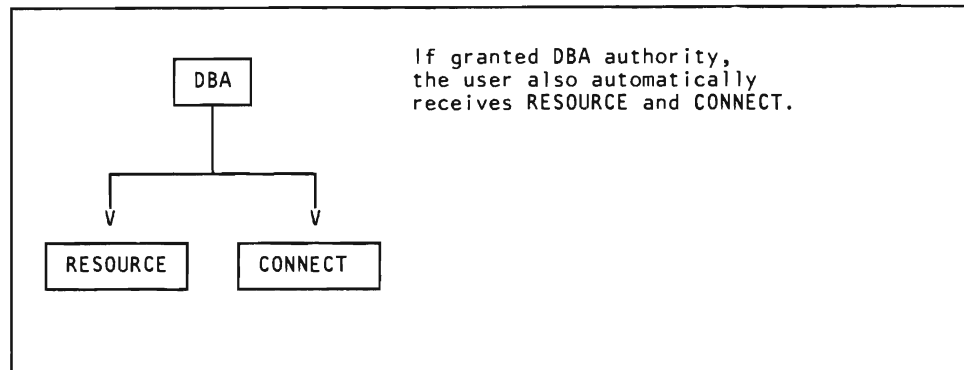


Figure 7. Hierarchy of SQL/DS Authority

Any holder of DBA authority may grant or revoke RESOURCE, CONNECT, or DBA authority to or from any other user. However, holders of RESOURCE authority without DBA authority cannot grant (or revoke) RESOURCE or CONNECT authority to (or from) other users.

DBA authority is the highest level of authorization provided in SQL/DS. If you have DBA authority, you are “immune” to the SQL/DS authorization mechanism; you can perform any operation on any table, or run any program. In addition, there are certain privileges that are available only to users with DBA authority. These privileges are listed in the *SQL/Data System Planning and Administration for VM/SP* manual.

If you hold DBA authority you may perform operations that are otherwise unauthorized, but you cannot grant or revoke these operations. For example, you may update the QUOTATIONS table even though you do not own this privilege explicitly, but you cannot grant or revoke this privilege unless you own it explicitly with the GRANT option. Similarly, if you preprocess a program containing some operation that you would not be authorized to perform except for your DBA authority, you receive RUN privilege on the program without the GRANT option. There is no entry in the SQL/DS catalog SYSPROGAUTH in this case.

The DBA functions are potentially dangerous to the integrity of the data base if misused. Therefore, an installation should carefully control the set of users who possess DBA authority, and a user with DBA authority should be very cautious in the use of those special powers. If you are granted DBA authority, you should read the *SQL/Data System Planning and Administration for VM/SP* manual.

The users who hold special privileges are listed in the SQL/DS catalog SYSUSERAUTH. Only users with DBA authority are allowed to access SYSUSERAUTH because the catalog contains userids and passwords. Other users can query the catalog through a view called SQLDBA.SYSUSERLIST. (Passwords are not seen in the view.)

Granting Privileges to Other Users

The GRANT statement allows you to pass privileges to other users. The most common and most convenient use of GRANT is via ISQL or the DBS utility. You can code GRANT statements within a program; however, because the userid and passwords in the GRANT statements can't be host variables, the statements have limited use. The GRANT statement has three formats:

Format 1 (for privileges on tables and views):

```
GRANT { [ ALTER
        DELETE
        INDEX
        INSERT
        SELECT
        UPDATE [(col-name-list)]
        ALL [PRIVILEGES] }
      ON [creator.] {table-name | view-name}
      TO { PUBLIC | userid1 [,userid2] ... } [WITH GRANT OPTION]
```

Note: ALTER, INDEX, and ALL [PRIVILEGES] do not apply to views.

Examples:

```
GRANT UPDATE (PARTNO, SUPPNO) ON QUOTATIONS TO SCOTT
GRANT SELECT, INSERT ON QUOTATIONS TO SMITH, JONES
GRANT ALL PRIVILEGES ON INVENTORY TO SCOTT WITH GRANT OPTION
```

Authorization:

You must possess the privilege with the GRANT option before you can grant that privilege to someone else.

Format 1 allows you to grant privileges on tables and views to other users. The *grantor* is considered to be the user who preprocessed the program in which this statement appears. (Certain exceptions to this rule are explained under "Dynamically Defined Statements" on page 147.) The *grantor* is considered to be the user who preprocessed the program in which this statement appears. (Certain

exceptions to this rule are explained under “Dynamically Defined Statements” on page 147.) A grant to PUBLIC is the same as a grant to all users.

The privileges you can grant are shown in Figure 8.

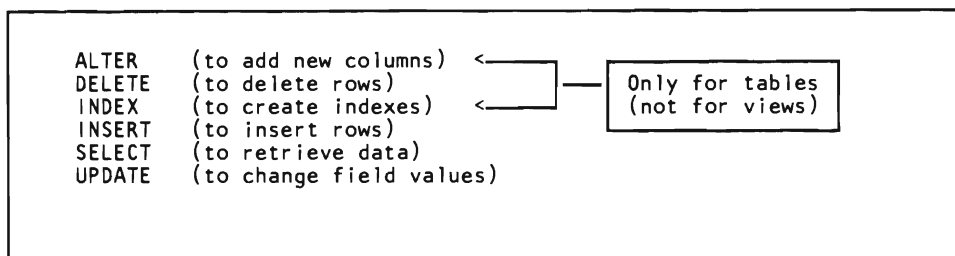


Figure 8. Privileges You Can Grant

Note especially that the ALTER privilege applies only to tables -- not to views or DBSPACES. (That is, it applies to the ALTER TABLE statement, but not the ALTER DBSPACE statement.)

You can specify more than one privilege. If you do, you can specify them in any order, but you must separate them with commas. To grant all six privileges, you can write ALL [PRIVILEGES] instead of listing all six. (Note that you can't grant ALL PRIVILEGES on a view; INDEX and ALTER privileges do not apply to views.) The PRIVILEGES keyword is both optional and non-functional; you can include it to improve readability. Thus, all of these statements are equivalent:

```
GRANT ALTER, DELETE, INDEX, INSERT, SELECT, UPDATE
ON QUOTATIONS TO SCOTT
```

```
GRANT DELETE, INDEX, ALTER, SELECT, UPDATE, INSERT
ON QUOTATIONS TO SCOTT
```

```
GRANT ALL PRIVILEGES
ON QUOTATIONS TO SCOTT
```

```
GRANT ALL
ON QUOTATIONS TO SCOTT
```

When you grant the UPDATE privilege on a table, you can optionally specify a list of column names. When you do, the grantee gets the power to update only those columns listed. If you choose not to specify a list of column names or if you specify ALL [PRIVILEGES], the grantee may update all columns of the table, even those created later via the ALTER TABLE statement.

If you specify WITH GRANT OPTION, the grantee may pass the granted privileges to other users.

Note that only the user who creates a table or view (or a user with DBA authority) can drop it. You can't grant a "drop" privilege to another user.

Before you grant privileges on views, you should read "Use of Views" on page 140.

Format 2 (for privileges on programs):

```
GRANT RUN ON [creator.]program-name  
  TO { PUBLIC | userid1 [,userid2] ... } [WITH GRANT OPTION]
```

Examples:

```
GRANT RUN ON TRANS1 TO EDWARDS WITH GRANT OPTION  
GRANT RUN ON JOB338 TO PUBLIC
```

Authorization:

You must possess the RUN privilege with the GRANT option before you can grant that privilege to someone else.

Format 2 allows you to grant privileges on programs to other users. The *grantor* is considered to be the user who preprocessed the program in which this statement appears. (Certain exceptions to this rule are explained under “Dynamically Defined Statements” on page 147.) A grant to PUBLIC is the same as a grant to all users.

The only privilege you can grant on a program is the RUN privilege, which lets another user run the indicated program. You can, however, pass on the RUN privilege with the GRANT option, just as you can with table privileges. The GRANT option permits the grantee to pass on that RUN privilege to others.

Note that only the user who preprocesses a program (or a user with DBA authority) can drop its access module from the data base. You can't grant a “drop” privilege to another user.

Format 3 (for special privileges):

```
GRANT { CONNECT
      { DBA
      { RESOURCE
      { SCHEDULE } } } TO userid1[,userid2...] [IDENTIFIED BY pass1[,pass2]]
```

Examples:

```
GRANT DBA TO BRUCE
GRANT CONNECT TO SMITH, JONES IDENTIFIED BY SECRET1, SECRET2
GRANT RESOURCE TO MARY, JIM, JOE
```

Authorization:

Generally, you must possess DBA authority to issue this statement. The exception is that can change your own password as explained below.

Format 3 allows a *user having DBA authority* to grant special privileges to other users. The *grantor* is considered to be the user who preprocessed the program in which this statement appears. (Certain exceptions to this rule are explained under “Dynamically Defined Statements” on page 147.)

The IDENTIFIED BY clause is optional when granting any of the special privileges. If the clause is included, a password is added or changed for each user specified. If the password is the same as currently exists for the user, the change has no real effect. If no passwords are given, none are assigned and previously assigned passwords are retained. If you have not been given a password, you cannot explicitly CONNECT to SQL/DS, but you may still have the capability of being implicitly connected. (See “VM/SP Connect Considerations” on page 186.)

Userids and passwords are limited to eight characters. They can be entered in double quotes to bypass checking under the rules of SQL identifier naming. (See “General Rules for Naming Data Objects” on page 74.) Embedded blanks are not permitted, even in double quotes. If you specify IDENTIFIED BY, you must include a password for every userid specified. The passwords and userids must correspond as indicated in the statement format above.

Granting any one of the special privileges to a user who does not already have the CONNECT authority causes that user to be granted CONNECT authority. For example, if a user currently has no special privileges and that user is granted RESOURCE authority, the user will have both RESOURCE and CONNECT authority.

A user can change his/her own password by using the GRANT CONNECT ... IDENTIFIED BY ... form of this command without requiring any special authority. To do this, the user need only have CONNECT authority, and may or may not have already been assigned a password.

Granting CONNECT to ALLUSERS is a special case that establishes implicit connect capability for all users in the system when operating under VM/SP. (See “VM/SP Connect Considerations” on page 186.)

Granting a special privilege that a user already possesses has no additional effect except for changing passwords if they are specified.

You should not grant CONNECT authority to SYSTEM or PUBLIC. They are used internally.

Note that a grant of SCHEDULE authority to a user is meaningless because SQL/DS allows only resource managers to use it.

Revoking Privileges from Other Users

The REVOKE statement allows you to take away the privileges of other users. (You can never revoke a privilege from yourself.) The most common and most convenient way to use REVOKE is via ISQL or the DBS utility. You can code REVOKE statements within a program; however, because the userid and passwords in the REVOKE statements can't be host variables, the statements have limited use.

If you attempt to revoke a privilege currently in use by a running program, the REVOKE statement is queued until the running program ends its current *logical unit of work*. Logical units of work are related groups of SQL statements (possibly with intervening host language code) that programmers define in their code. Thus, if you revoke the UPDATE privilege from user MARY, but MARY's program is running and is already making updates, your REVOKE statement won't take effect until MARY's updates are finished. Logical units of work are discussed more completely under “Data Control” on page 70.

When you revoke a privilege on a table, view, or program from a user X, SQL/DS automatically revokes it from all users to whom X has granted it, unless they have some other source for the privilege that is not dependent on user X.

Special privileges, which can be granted and revoked only by a user with DBA authority, are handled slightly differently. That is, if you have DBA authority, and revoke a special privilege (such as RESOURCE authority) from a user X, no other users are affected. In addition, if a user with DBA authority revokes RUN authority from user X, no other users are affected. (The “cascade” effect described earlier for the RUN privilege does not apply to users with DBA authority.)

In some cases, SQL/DS automatically revokes the RUN privilege from a number of users. Suppose a user GENE has preprocessed a program that makes use of some privilege, such as SELECT. GENE receives the RUN privilege on the program with the GRANT option, and perhaps he may grant this privilege to other users. If the SELECT privilege is now revoked from GENE, the access module associated with the program is automatically marked invalid. When the program is next run (by GENE or by any other user), SQL/DS attempts to regenerate a valid (fully authorized) access module. At the time of this regeneration process, the following outcomes are possible:

1. GENE has all the privileges required by the program, and furthermore has the GRANT option on all these privileges. In this case, the access module is regenerated, all existing grants of the RUN privilege on the program remain in effect, and execution proceeds normally.
2. For some SQL statement in the program, GENE lacks the necessary privilege, or has the privilege without the GRANT option. In this case, GENE retains the RUN privilege on the program, but all existing grants of the RUN privilege are revoked. When the program is run, those SQL statements for which GENE has the necessary privilege execute successfully, and other SQL statements return error codes.

SQL/DS can also automatically revoke privileges on views or drop the view definition. Suppose BILL grants GENE the SELECT privilege with the GRANT option on the EMPLOYEES table. GENE then defines a view called SALARY on the EMPLOYEES table, and grants the SELECT privilege on that view to other users. After some time, BILL decides to revoke the SELECT privilege on the EMPLOYEES table from GENE. When BILL revokes the SELECT privilege on the table, SQL/DS automatically revokes the SELECT privilege from SALARY also, including all SELECT privileges on SALARY that GENE passed on. If, after this process, GENE holds no privileges on SALARY, the definition of SALARY is dropped from SQL/DS.

The REVOKE statement has three formats:

Format 1 (for privileges on tables and views):

```

REVOKE {
  [ ALTER
    DELETE
    INDEX
    INSERT
    SELECT
    UPDATE ]
  ALL [ PRIVILEGES ]
} ON [creator.] {table-name | view-name}
FROM { PUBLIC | userid1 [,userid2] ... }

```

Note: ALTER, and INDEX, do not apply to views. ALL [PRIVILEGES] does apply, however. (See following text.)

Examples:

```

REVOKE SELECT, INSERT ON QUOTATIONS FROM SMITH, JONES
REVOKE UPDATE ON INVENTORY FROM PUBLIC
REVOKE ALL ON SUPPLIERS FROM SCOTT

```


Authorization:

You can revoke only those privileges you have granted to other users, not those another user has granted.

Format 1 allows you to revoke privileges you have granted on tables and views. The *revoker* is considered to be the user who preprocessed the program in which this statement appears. (Certain exceptions to this rule are explained under “Dynamically Defined Statements” on page 147.) When you revoke authority from PUBLIC, SQL/DS revokes the indicated privileges you have explicitly granted to PUBLIC (via GRANT ... TO PUBLIC). It does *not* revoke all your grants of the indicated privilege. For example, if you grant UPDATE on QUOTATIONS to SMITH, JONES, and PUBLIC, and then revoke this privilege from PUBLIC, the privilege is still held by SMITH and JONES.

You can specify more than one privilege that you wish to revoke. If you do, you can specify them in any order, but you must separate them with commas. If you specify ALL [PRIVILEGES] instead of listing the privileges, all table (or view) privileges you have granted to the indicated user(s) are revoked. You can use ALL [PRIVILEGES] even if you have not granted all six table privileges to the user. “REVOKE ALL PRIVILEGES” means “revoke all table privileges granted by this grantor to this grantee,” regardless of whether the grantee has a complete list of privileges. The revokee still retains any privileges obtained from another source. The PRIVILEGES keyword is optional and non-functional; you can include it to improve readability.

Recall that in the GRANT statement you can specify a list of columns when you granted the UPDATE privilege. When revoking an UPDATE privilege, you cannot list specific columns for which you want to revoke the privilege. “REVOKE UPDATE” means “revoke all those update privileges granted by this grantor to this grantee” (regardless of whether you originally specified a column list when you granted the privilege).

Note that the only way to revoke the GRANT option on a privilege is to revoke the privilege itself. (Of course, you can then re-grant the privilege without the GRANT option.)

Format 2 (for privileges on programs):

```
REVOKE RUN ON [creator.]program-name FROM { PUBLIC | userid1 [,userid2] ... }
```

Example:

```
REVOKE RUN ON TRANS1 FROM SMITH
```

Authorization:

You can revoke the RUN privilege from only those users to whom you have granted it.

Format 2 allows you to revoke the RUN privilege you have granted on programs. The *revoker* is considered to be the user who preprocessed the program in which this statement appears. (Certain exceptions to this rule are explained under “Dynamically Defined Statements” on page 147.) When you revoke the RUN privilege from PUBLIC, SQL/DS revokes the privilege you have explicitly granted to PUBLIC (via GRANT RUN ON ... TO PUBLIC). It does *not* revoke all your grants of the RUN privilege. For example, if you grant RUN on TRANS1 to SMITH, JONES, and PUBLIC, and then revoke this privilege from PUBLIC, users SMITH and JONES are still able to run the TRANS1 program.

If you have granted the RUN privilege with the GRANT option, the only way to revoke the GRANT option is to revoke the RUN privilege itself (of course, you can then re-grant the RUN privilege without the GRANT option).

Format 3 (for special privileges):

```
REVOKE {CONNECT | RESOURCE | SCHEDULE | DBA} FROM userid1 [,userid2] ...
```

Example:

```
REVOKE DBA FROM SMITH
```

Authorization:

You must possess DBA authority to issue this statement.

Format 3 allows a *user having DBA authority* to revoke special privileges from other users. The *revoker* is considered to be the user who preprocessed the program in which this statement appears. (Certain exceptions to this rule are explained under “Dynamically Defined Statements” on page 147.)

A user with DBA authority may revoke any special privilege from any user, regardless of who originally granted the privilege. The only two exceptions are:

1. A user having DBA authority cannot revoke any authority from himself, and
2. No one can revoke RESOURCE authority from a user that has DBA authority.

If you issue REVOKE for a special privilege that the user doesn't have, the revocation is ignored for that privilege. Revoking CONNECT causes all special privileges to be revoked with it and the user is deleted from the SQL/DS catalog SYSUSERAUTH. Revoking CONNECT does *not* cause objects owned by that user to be dropped. A user with DBA authority can drop them.

Revoking DBA authority automatically causes all special privileges to be revoked except CONNECT. Revoking RESOURCE authority implies no other revocations.

Data Control

SQL Data Control statements manage *DBSPACES*, which are units of space, and logical units of work, which are sequences of SQL statements that SQL/DS treats as a single entity.

How the Data Base Is Structured

A *DBSPACE* is a portion of the data base that can contain one or more tables and any associated indexes. Each table stored in SQL/DS is placed in some particular *DBSPACE* chosen by the creator of the table.

DBSPACES are defined when the data base is generated and may be added later via the ADD *DBSPACE* function. Each *DBSPACE* remains as an unnamed “available” *DBSPACE* until it is “acquired” by means of an ACQUIRE *DBSPACE* statement.

The user who acquires the *DBSPACE* (generally the DBA) may specify a *storage pool* from which SQL/DS is to acquire the *DBSPACE*, or may allow SQL/DS to choose the storage pool by default. A storage pool is a collection of data sets called *DBEXTENTS*. Storage pools are numbered from 1 to 999. They allow for the controlling of the distribution of the data base across *DASDs*.

Storage pools can be recoverable or non-recoverable. Recoverable storage pools protect their data using the SQL/DS automatic recovery for data updates. “Non-recoverable” means that if there is a system failure, some data may be lost. But, when data is non-recoverable, system overhead is reduced. See the *SQL/Data System Planning and Administration for VM/SP* manual for more information about storage pools.

The acquiring user gives a name to the *DBSPACE*, and defines certain characteristics for it. If the type of *DBSPACE* is PRIVATE, the user who acquired it becomes its *owner*; if it is type PUBLIC, its owner becomes PUBLIC. An acquired *DBSPACE* may later be returned to the list of available *DBSPACES* by the DROP *DBSPACE* statement.

A user holding RESOURCE authority may create new tables in any PUBLIC *DBSPACE*, or in any PRIVATE *DBSPACE* owned by that user. A user who does not have RESOURCE authority may also create tables in any PRIVATE *DBSPACE* that was acquired for that user by the DBA. Only users having DBA authority can create tables in a PRIVATE *DBSPACE* owned by another user.

Your ability to access and update tables of another user is controlled by SQL/DS. If you are authorized, you can access and update tables in any *DBSPACE* of any type, (except for SQL/DS catalogs, which you can read but not update). To refer

to a table created by another user, use the creator's userid as a prefix to the table name (for example, SMITH.INVENTORY).

Even though you may be authorized to access data in someone else's DBSPACE, you may not be permitted to access the data if the DBSPACE is in use.

An attempt to *read* data in a PRIVATE DBSPACE results in a negative SQLCODE if any data in the DBSPACE has been *modified* by a still-active logical unit of work. An attempt to *modify* data in a PRIVATE DBSPACE results in a negative SQLCODE if any data in the DBSPACE has been *read or modified* by a still-active logical unit of work. Note the difference between the two DBSPACE-types in terms of their locking behavior. If you attempt to access locked data in a PUBLIC DBSPACE, your program waits and does not regain control until the lock is freed. (This waiting period is "transparent" to your application program.) If you attempt to update locked data in a PRIVATE DBSPACE, SQL/DS immediately returns control to your program with a negative SQLCODE.

The size of the space that is locked is called the *lock size*. The lock size on a PRIVATE DBSPACE is always the entire DBSPACE. The default lock size on a PUBLIC DBSPACE, however, is somewhat smaller to allow more concurrency. Thus, you should place tables in a PUBLIC DBSPACE if you expect that more than one user may need concurrent access to them. On the other hand, because operations on PRIVATE DBSPACES do not pay the overhead of acquiring individual locks within the DBSPACE, a PRIVATE DBSPACE is an efficient place to store tables for the exclusive use by one user at a time. The cost of smaller locks is higher overhead. Figure 9 and Figure 10 summarize the SQL/DS locking mechanism.

If you attempt to:	But another user has already:	
	read the data (acquired a share lock)	modified the data (acquired an exclusive lock)
read data	You are allowed to read the data.	You receive a negative SQLCODE.
modify data	You receive a negative SQLCODE.	You receive a negative SQLCODE.
The lock size for a PRIVATE DBSPACE is always the entire DBSPACE.		

Figure 9. Locking Summary for PRIVATE DBSPACES

If you attempt to:	But another user has already:	
	read the data (acquired a share lock)	modified the data (acquired an exclusive lock)
read data	You are allowed to read the data.	Your program waits.
modify data	Your program waits.	Your program waits.
The lock size of a PUBLIC DBSPACE defaults to a page (4096 bytes). The lock size can be changed by the ACQUIRE DBSPACE or ALTER DBSPACE statements.		

Figure 10. Locking Summary for PUBLIC DBSPACES

Logical Units of Work

A *logical unit of work* is a sequence of SQL statements (possibly with intervening host language code) that SQL/DS treats as a single entity.

SQL/DS ensures the consistency of data at the logical unit of work level. That is, SQL/DS ensures that either *all* operations within a logical unit of work are completed, or *none* of them are completed. Suppose, for example, that money is to be deducted from one account and added to another. If both these updates are placed in a single logical unit of work, the data will not be left in an inconsistent state. If a failure of the system or a user program occurs while a logical unit of work is in progress, the data is automatically restored to its state before the logical unit of work began. (When a program fails, SQL/DS restores the data after detecting the error. See "Error Handling" on page 202 for more information. When the system fails, the data is restored when SQL/DS is restarted.)

A logical unit of work is begun implicitly when the first *executable* SQL statement is encountered (except a CONNECT statement). A logical unit of work is ended by either a COMMIT WORK or ROLLBACK WORK statement. If there is no COMMIT or ROLLBACK WORK, the logical unit of work ends when the program ends.

These SQL declarative statements do not start a logical unit of work:

```
BEGIN DECLARE SECTION          DECLARE CURSOR
END DECLARE SECTION           INCLUDE SQLCA
WHENEVER                      INCLUDE SQLDA
```

Executable SQL statements always occur within a logical unit of work. This is because any executable SQL statement (except CONNECT) encountered after you end a logical unit of work automatically starts another.

The ROLLBACK WORK statement described later allows a program to explicitly call for the restoring of the logical unit of work and associated data.

Dropping a Program

Format:

```
DROP PROGRAM [creator.]program-name
```

Examples:

```
DROP PROGRAM PAYROLL2
DROP PROGRAM SALLY.RUNRUN
DROP PROGRAM :CREATOR.:PROGNAME
```

Authorization:

You can only drop programs that you have preprocessed. (That is, you must be the creator of the program you wish to drop.) To drop another user's program, you must have DBA authority.

The DROP PROGRAM statement deletes the access module associated with the named program from the data base. Once you drop an access module, you cannot run the program.

If a running program drops its own access module, it receives a negative return code when it attempts to begin the next logical unit of work.

To re-create an access module, preprocess the program. Once the access module is created, you'll be able to run the program.

You can specify both the creator and program-name as host variables (fixed length, eight characters, padded to the right with blanks) or as constants. If host variables are used, you can provide either value at the time the program is run.

Note: The program-name is the name specified in the PREPNAME parameter when the program is preprocessed. If the program name is an SQL/DS reserved word, you must enclose it in double quotes (") when used in the DROP PROGRAM statement. When used in the PREPNAME preprocessor parameter, however, the name should *not* be enclosed in double quotes. For example, when preprocessing a program you can specify:

```
PREPNAME=SELECT
```

When dropping that program, however, you must specify:

```
DROP PROGRAM "SELECT"
```

See "Preprocessing and Running the Program" on page 183 in Chapter 2 for more information about preprocessor parameters. Appendix A, "SQL/DS Reserved Words" on page 363 contains a list of SQL/DS reserved words.

Data Definition

SQL Data Definition statements manage tables and things you can associate with tables (such as indexes, synonyms, and comments).

General Rules for Naming Data Objects

In general, the following SQL identifiers must conform to specific naming rules:

1. Table names
2. View names
3. Column names
4. Index names
5. Synonyms
6. DBSPACE names
7. Program names
8. Cursor names
9. Statement names
10. Host variable names
11. Userids/creator names
12. Passwords

Folding from lowercase to uppercase is always performed for identifier types 1 through 7 above, as long as the identifiers are not enclosed in quotes.

The naming rules are:

1. A name may begin with an uppercase letter (A-Z), \$, #, or @. A name may begin with a number (0-9) if it is enclosed in double quotes.
2. It may contain uppercase letters (A-Z), \$, #, @, numbers (0-9), or underscores (_).

As a general rule, the length is 1-18 characters. Exceptions are as follows:

1. Program names are 1-8 characters.
2. Userids and passwords: Constants are 1-8 characters; host variables are 8 characters, padded to the right with blanks when the value is less than 8. Lowercase characters, special characters, and Double-Byte Character Set (DBCS) characters should not be used in SQL userids or passwords.
3. Host variables are limited to 18 characters, unless the host language has a lesser restriction. Note that FORTRAN permits only six-character host variable names. Examples in this manual sometimes exceed the FORTRAN limitation.

When identifiers are stored in host variables and then referred to in SQL statements, they are generally treated by SQL/DS as if they were entered in double quotes. That is, the general identifier rules are not checked when they appear in host variables. An example is the CONNECT statement, where the userid (and

password) must be in host variables. The host variable(s) in this case may contain any eight characters.

Some exceptions to the identifier naming rules should be noted.

1. Where the host language has restrictions on variable names, those rules will further restrict the SQL naming rules as applied to host variable names. In COBOL, host variables may contain dashes (-) in lieu of the underscore (_).
2. Generally, SQL reserved words cannot be used as data object names. These are listed in Appendix A.
3. For SQL/DS installations that use an EBCDIC character set other than English, the two naming rules and the folding rule stated above may change slightly. Refer to the *SQL/Data System Planning and Administration for VM/SP* for more information.

The above rules for naming may be bypassed in most cases by including the name in double quotes. In this way, lowercase letters, special characters, blanks, and reserved words may be used in identifiers. For example,

```
"quotations"      "DAVE'S TABLE"      "SELECT COLUMN"
```

You cannot use the double quote (") character within a double quoted string:

```
"EMP"13"TABLE"      <----- Not Valid
```

Leading blanks cannot be used in double quoted strings. If they are, an error will result.

```
" TABLEX"          <----- Not Valid
```

There are a few cases that do not permit use of double quoted identifiers:

1. Host variable names
2. Program names
3. Cursor names
4. Statement names.

If you use Double-Byte Character Set data, and if the DBCS option is set to YES, both unquoted and quoted identifiers can have DBCS characters enclosed by `so` and `si`. The length limit (in bytes) applies to the total length of EBCDIC portions, DBCS portions and shift characters. The folding rule does not apply to the DBCS portions. With the DBCS option set to YES, an apostrophe (X'7F') in a DBCS character does not terminate a quoted identifier.

Data Types

Each column of every SQL/DS table is given an SQL data type when the table is created. There are ten of these SQL/DS data types. The data types for Double-Byte Character Set (DBCS) data support character sets that require two bytes of storage for each character in the character set. Kanji is one example of such a character set. Figure 11 shows the ten SQL/DS data types and how they are stored internally:

SQL/DS Data Type	How Stored
INTEGER	Stored as a 31-bit binary integer.
SMALLINT	Stored as a 15-bit binary integer.
DECIMAL(m,[n])	Stored as a packed decimal number of precision m and scale n. Precision is the total number of digits. Scale is the number of digits to the right of the decimal point. For example, 251.66 fits in a DECIMAL(5,2) data area. The default scale is 0. If an even value is specified for m, SQL/DS rounds that value to the next higher odd value to best utilize internal storage.
FLOAT	Stored as a double-precision (8-byte) floating-point number in standard System/370 floating-point format.
CHAR(n)	Stored as an EBCDIC character string of fixed length n. (n cannot be larger than 254.)
VARCHAR(n)	Stored as a varying-length EBCDIC character string of maximum length n. (n cannot be larger than 254.)
LONG VARCHAR	Stored as a varying-length EBCDIC character string of maximum length 32767.
GRAPHIC(n)	Stored as a Double-Byte Character Set (DBCS) character string of fixed length n. (n cannot be larger than 127.)
VARGRAPHIC(n)	Stored as a varying-length DBCS character string of maximum length n. (n cannot be larger than 127.)
LONG VARGRAPHIC	Stored as a varying-length DBCS character string of maximum length 16383.

Figure 11. SQL/DS Data Types

Data Conversion

Whenever SQL/DS moves data from one field to another, or from a host variable to a field, or from a field to a host variable, it attempts to perform data conversion if the data types do not match. Whether or not the conversion is successful depends on the data types of *source* value and the *target* value.

For example, suppose you issue a SELECT statement that retrieves INTEGER data (source data) into a host variable that was declared SMALLINT (target variable). If the INTEGER value is small enough, SQL/DS performs the operation successfully. If the INTEGER value is larger than the largest value that can fit in a SMALLINT variable, however, an overflow results and SQL/DS indicates a conversion error by returning a negative SQLCODE.

SQL/DS data conversion is summarized in Figure 12. YES indicates that SQL/DS does the conversion. NO indicates that the conversion is not done, and SQL/DS returns an error code to your program. Notice that overflow (loss on the left) or truncation (loss on the right) may occur on some conversion attempts.

Overflows always cause an SQL error (negative SQLCODE). Truncations are handled differently for numeric and character data:

- Numeric data: Truncation of zeros on the left or truncation of the fractional part of decimal or floating point values takes place without error or warning. Any other loss of data on conversion is considered an overflow error.
- Character Data (EBCDIC and DBCS): When output from SQL/DS does not fit into a host variable, a warning condition exits. SQLWARN1 is set to indicate truncation. In this case, if you provide an indicator variable, the value within it denotes the actual length of the variable in characters before truncation. Indicator variables are discussed under “Indicator Variables” on page 146.

When an input character string value does not fit into an SQL/DS field, an error results.

TARGET DATA TYPE.

SOURCE DATA TYPE:	INTE-GER	SMALL-INT	DEC-IMAL	FLOAT	CHAR ³	VAR-CHAR ³	LONG VAR-CHAR ⁴	GRAPHIC ³	VAR-GRAPHIC ³	LONG VAR-GRAPHIC ⁴
INTEGER	YES	YES ¹	YES	YES	NO	NO	NO	NO	NO	NO
SMALLINT	YES	YES	YES	YES	NO	NO	NO	NO	NO	NO
DECIMAL	YES ^{1 2}	YES ^{1 2}	YES ⁵	YES ⁶	NO	NO	NO	NO	NO	NO
FLOAT	YES ^{1 2}	YES ^{1 2}	YES ^{5 6}	YES	NO	NO	NO	NO	NO	NO
CHAR	NO	NO	NO	NO	YES	YES	YES	NO	NO	NO
VARCHAR	NO	NO	NO	NO	YES	YES	YES	NO	NO	NO
LONG VARCHAR	NO	NO	NO	NO	YES	YES	YES	NO	NO	NO
GRAPHIC	NO	NO	NO	NO	NO	NO	NO	YES	YES	YES
VAR-GRAPHIC	NO	NO	NO	NO	NO	NO	NO	YES	YES	YES
LONG VAR-GRAPHIC	NO	NO	NO	NO	NO	NO	NO	YES	YES	YES

Figure 12. SQL/DS Data Conversion Chart

Notes:

1. Overflow may result.
2. The fractional part of the value is dropped.
3. For output host variables, if the length of the target is smaller than the length of the source, truncation occurs and SQLWARN1 is set. If an indicator variable is given for an output value, it is set to the actual SQL/DS field length. For input host variables that exceed the length of the target field, an error results.
4. Note the restrictions under “Use of Long Fields” on page 236.

5. SQL/DS automatically aligns the decimal point. Overflow of the integer part may result. The fractional part may be truncated.
6. SQL/DS attempts to create the “best possible” result in converting from System/370 floating point to scaled fixed point decimal.

If you need more information about how computations are performed internally, or how overflows can occur, refer to the “Arithmetic Operations” section of the *SQL/Data System Planning and Administration for VM/SP* manual.

Qualifying Table Names

If a data object (such as a table) is owned by another user, you need to qualify references to the object by concatenating the creator’s user identifier:

```
SMITH.INVENTORY
-----
|           |
|           |-----> table name
|           |-----> creator of the table
```

The period (.) is the SQL/DS concatenation symbol.

You can access another user’s table only if you know that person’s user identifier and have the appropriate SQL/DS authorization to access that table.

When you concatenate a userid to a table name, you *fully qualify* the table. A table is fully qualified when a userid is concatenated to it. **That is, a “userid.table-name” uniquely identifies a table in the data base.** For example, there can never be two SMITH.INVENTORY tables in the data base at the same time.

You should use fully qualified table names until you gain some experience using SQL/DS. By fully qualifying table names, you avoid confusion and errors. This is especially true if you are coding programs that are to be preprocessed by another user.

SQL/DS Catalogs

SQL/DS automatically maintains information about the data base in a set of tables called *catalogs*. These catalogs are created automatically during data base generation. They describe tables, columns, indexes, programs, authorization, and other objects in the data base.

Since the SQL/DS catalogs are defined as normal tables with public read authorization, you can use SQL query statements to retrieve information in the catalogs. For example, this SQL statement finds what column names in table EMP TABLE begin with the letter ‘D’:

```
SELECT CNAME
FROM SYSTEM.SYSCOLUMNS
WHERE TNAME = 'EMP TABLE'
AND CNAME LIKE 'D%'
```

Note that when a table name is used as a constant, it is enclosed in single quotes ('), not double (").

SYSTEM is the owner of all catalog tables; you must qualify all catalog tables with that name, unless you have a synonym defined.

The only information in the tables not available to everyone is password information; you must have DBA authority to access the catalog that contains passwords (SYSUSERAUTH). A view, called SYSUSERLIST, is defined on SYSUSERAUTH when the catalogs are created. The creator of the view is SQLDBA; thus, you must refer to the view as SQLDBA.SYSUSERLIST. This view is accessible to all users and contains all the columns of SYSUSERAUTH except the passwords. If you do not have DBA authority, you must query the view (SYSUSERLIST) instead of the underlying table (SYSUSERAUTH).

Some of the information in the catalogs is of little interest to most users. Statistics maintained in the catalogs, for example, are used by SQL/DS to determine optimal access paths -- to you, these statistics may be quite meaningless. If you wish, you can define views on the catalog tables containing only columns that are meaningful to you.

SQL/DS updates its catalogs during normal operation in response to SQL data definition and control statements. Additionally, if you have DBA authority you can create and maintain your own installation-dependent catalog columns using SQL INSERT, DELETE, UPDATE, ALTER, and COMMENT statements.

The catalogs are completely described in the *SQL/Data System Planning and Administration for VM/SP* manual. A brief description of each catalog is given below.

Catalogs that Record Privileges

SYSUSERAUTH

SQL/DS uses SYSUSERAUTH to record special privileges. The special privileges are DBA, RESOURCE, SCHEDULE, and CONNECT authority. As in SYSTABAUTH, an entry in SYSUSERAUTH denotes either a special privilege held by a user or a special privilege exercised by a program.

SYSUSERLIST

Only users with DBA authority can access SYSUSERAUTH; other users must access the view SYSUSERLIST. The creator of the view is SQLDBA; thus, you must refer to the view as SQLDBA.SYSUSERLIST. The SYSUSERLIST view contains all columns of SYSUSERAUTH except PASSWORD.

SYSPROGAUTH

SYSPROGAUTH records privileges of users to run programs, and to grant these privileges to other users.

SYSTABAUTH

SYSTABAUTH has two purposes:

1. It records privileges owned by users to access tables and views. For each such privilege, it also records the source of the privilege (for example, a grant from another user).
2. It records the privileges on tables and views that are exercised by various preprocessed programs. Each such privilege appears in SYSTABAUTH as if it were *granted* to the program by the user who preprocessed the program. SQL/DS uses this type of SYSTABAUTH entry to find and invalidate access modules when the necessary privileges are revoked from the creators of the program.

SYSCOLAUTH

SYSCOLAUTH records grants of the UPDATE privilege on tables and views when the privilege is granted on a column-by-column basis. Each entry in SYSCOLAUTH has a corresponding entry in SYSTABAUTH with a matching timestamp. (SYSTABAUTH records privileges granted on entire tables, but not on individual columns.) A SYSCOLAUTH entry identifies a particular column on which an UPDATE privilege has been granted. For example, if the UPDATE privilege is granted on several columns in one GRANT statement, the grant is represented as one entry in SYSTABAUTH, and several entries in SYSCOLAUTH, all having matching timestamps.

Some of the entries in SYSCOLAUTH represent privileges that are exercised by preprocessed programs. These entries appear as though the creator of the program (that is, the user who preprocessed the program) granted the privilege to the program itself.

Catalogs that Record the Contents of the Data Base

SYSDBSPACES

The SYSDBSPACES catalog contains a row for each DBSPACE in the data base, including those DBSPACES that no user has yet acquired. The number of DBSPACES available is determined during data base generation. The size of each DBSPACE is also specified at that time.

Additional DBSPACES may be added from time to time by the ADD DBSPACE utility programs; refer to the *SQL/Data System Planning and Administration for VM/SP* manual for more information about these utilities.



SYSCATALOG

The SYSCATALOG table contains a row for each table or view in the data base, including itself and other catalog tables.

SYSACCESS

SQL/DS uses SYSACCESS to record the access modules that have been created for user programs by the SQL/DS preprocessor. Some entries in SYSACCESS are also used to record view definitions in a form for internal use.

SYSVIEWS

The SYSVIEWS catalog contains the definitions of all views known to SQL/DS. The views are stored in the form of the original SQL statements that defined the views.

SYSCOLUMNS

The SYSCOLUMNS catalog contains a more detailed description of the data base than that contained in SYSCATALOG. Recall that SYSCATALOG contains a row for each table or view in the data base; SYSCOLUMNS contains a row for every *column* of every table or view in the data base (including the columns of the SQL/DS catalogs).



Catalogs that Record Indexes and Synonyms

SYSINDEXES

The SYSINDEXES catalog contains a row for every index currently in existence, including the indexes that SQL/DS maintains on its own catalogs.

SYSSYNONYMS

The SYSSYNONYMS catalog contains a row for every synonym that is currently in effect. Note that each synonym is effective for *only* the user who defined it.

Miscellaneous Catalogs

SYSUSAGE

SYSUSAGE records dependencies of one SQL/DS object on another. For example, an access module is dependent on the tables and indexes that it uses, or a view is dependent on the tables on which it is defined. Each entry in SYSUSAGE describes one dependent object and one base object. (The base object is the object that is depended upon.)

SYSDROP

This catalog forms part of the mechanism used by SQL/DS to drop tables and DBSPACES from the data base. When a DBSPACE or table is dropped, its description is dropped from the SQL/DS catalogs immediately, but the underlying Data Base Storage System (DBSS) objects are not dropped until the end of a logical unit of work. (The DBSS is an internal component of SQL/DS.) SYSDROP contains a list of the DBSS objects that are waiting to be dropped.

SYSCHARSETS

Contains a column for the EBCDIC character classification table (to identify valid characters) and a column for the EBCDIC character translation table (for folding to uppercase).

SYSOPTIONS

Records whether or not Double-Byte Character Set data can be used for identifiers and character string constants. Also records what EBCDIC character set SQL statements are written in.

Chapter 2. Advanced SQL Programming

This chapter builds off the information contained in Chapter 1. For each of the five tasks described in Chapter 1, a corresponding section exists in this chapter, containing more detailed information.

The first section of this chapter, “Designing the Program,” contains a detailed explanation of the framework for coding SQL application programs. This framework was introduced in Chapter 1. This section also describes the SQL statements that must be included in the prolog and epilog sections of the program.

The second section, “Coding the Program,” describes advanced SQL statements and clauses that you might wish to use in your programs.

The third section of this chapter, “Preprocessing and Running the Program,” gives detailed descriptions of the steps necessary to run your program.

The fourth section, “Testing and Debugging Concerns,” tells you how to handle errors that arise during the execution of your program.

The fifth section, “Putting the Program into Production,” contains advanced information that may be useful to you from an administrative standpoint.

Because this chapter contains advanced information, and because most of the topics are self-contained, there are no section quizzes to determine whether you need to read the sections.

Designing the Program

Contents

Application Prolog	86
Declaring the SQLCA	86
Host Variables	87
Connecting to SQL/DS	91
Application Body	92
Application Epilog	93
CMS Applications	93
Summary	94
Sample Application Programs	95

Application Prolog

At the beginning of every SQL/DS program, you must place SQL statements that:

- Declare a SQL Communications Area (SQLCA) and provide for error handling
- Declare special variables (host variables) that SQL/DS uses to interact with the host program
- Establish a connection between your program and SQL/DS.

Declaring the SQLCA

To declare the SQL Communications Area (SQLCA), code this statement in your program:

```
INCLUDE SQLCA
```

When you preprocess your program, SQL/DS inserts host language variable declarations in place of the INCLUDE SQLCA statement. This group of variables is how SQL communicates with your program. SQL/DS uses the variables for warning flags, error codes and diagnostic information. All the variables are discussed under “Testing and Debugging Concerns” on page 201. The only variable you need be concerned with now is SQLCODE.

SQL/DS returns a result code in SQLCODE after executing each SQL statement. SQLCODE, return code, and result code are all terms that mean the same thing: the integer value that summarizes how your SQL statement executed. When a statement executes successfully, SQLCODE is set to 0. SQL/DS indicates error conditions by returning a negative SQLCODE. A positive SQLCODE indicates normal conditions experienced while executing the statement (such as end-of-file).

The WHENEVER statement below tells SQL/DS what to do when it encounters an SQL error (that is, a negative SQLCODE):

```
WHENEVER SQLERROR GO TO ERRCHK
```

That is, whenever an SQL error (SQLERROR) occurs, program control is transferred to a subroutine named ERRCHK. This subroutine should include logic to analyze the error indicators in the SQLCA. Depending upon how ERRCHK is defined, action may be taken to execute the next sequential program instruction, to perform some special functions, or, as in most cases, to roll back the current logical unit of work and terminate the program.

You can have any number of logical units of work in a program. For the simplest case (which is being discussed here) the whole program is a single logical unit of work. Either the program runs successfully and the changes are made to the data base, or it doesn't and no changes are made.

SQL/DS begins a logical unit of work implicitly. That is, you don't have to code a statement to start a logical unit of work. SQL/DS starts one when it encounters

your first executable SQL statement. (“Logical Units of Work” on page 72 gives a more precise description of when logical units of work begin.)

You must tell SQL/DS when to end the logical unit of work. “Application Epilog” on page 93 explains how to do this. There are times when SQL implicitly ends a logical unit of work. When this occurs, the SQLWARN0 and SQLWARN6 indicators are set to ‘W’.

Host Variables

You must declare all host variables. In addition, you must surround the host variable declarations with two SQL statements:

```
BEGIN DECLARE SECTION
  .
  .
  (host variable declarations)
  .
  .
END DECLARE SECTION
```

The data declaration statements vary from language to language. To determine what the data declarations should be, you need to be familiar with SQL/DS data types.

Consider the following SELECT statement:

```
SELECT DESCRIPTION, QONHAND
INTO :DESC, :QUANT
FROM INVENTORY
WHERE PARTNO = :PART
```

The statement contains three host variables: DESC, QUANT, and PART. The host variables interact with columns of the SQL/DS INVENTORY table. Each column of every SQL/DS table is given an SQL data type when the table is created. There are ten of these SQL/DS data types. Figure 11 on page 76 shows the ten SQL/DS data types and how they are stored internally.

Each SQL/DS data type can be related to a host language data type. For example, the INTEGER SQL/DS data type is a 31-bit binary integer. In COBOL this is equivalent to a data description entry of:

```
01 variable-name PICTURE S9(9) COMPUTATIONAL.
```

In PL/I, the INTEGER data type equates to:

```
DCL variable-name BINARY FIXED(31)
```

In FORTRAN, this equates to:

```
INTEGER variable-name
```

And, in Assembler:

```
variable-name DS F
```

All the host language equivalents for a particular SQL/DS data type are listed in the host language appendixes. The charts are at the end of each host language appendix. See Figure 40 on page 383, Figure 42 on page 417, Figure 46 on page 446, or Figure 49 on page 464.

It is a simple matter to see which host variables interact with which columns. Here is the SELECT statement again:

```
SELECT DESCRIPTION, QONHAND
INTO :DESC, :QUANT
FROM INVENTORY
WHERE PARTNO = :PART
```

The DESCRIPTION column of the selected row is returned in DESC. The QONHAND column is returned in QUANT. The PARTNO column is compared to the PART host variable.

Once you determine which column a host variable interacts with, you need to find what SQL/DS data type that column has. The SQL/DS data types for the example table are listed in the upper right hand corner of the foldout (along with the tables). DESCRIPTION is VARCHAR(24), QONHAND is INTEGER, and PARTNO is SMALLINT. (For now, you can ignore the "NOT NULL" in the chart.)

When you are coding an actual program, you can find out what data type a given column has by asking the person who created the table. Alternatively, you can query the SQL/DS catalogs. The catalogs are tables maintained by SQL/DS. They contain information about all the tables created in the data base. The catalogs are completely described in the *SQL/Data System Planning and Administration for VM/SP* manual.

Having determined the SQL/DS data types, you can refer to the conversion charts at the end of the host language appendixes and code the appropriate declarations. Figure 13 shows the declarations in each host language.

COBOL	<pre> Cols. 8 12 EXEC SQL BEGIN DECLARE SECTION END-EXEC. 01 DESC. 49 D-LENGTH PICTURE S9(4) COMPUTATIONAL. 49 D-VALUE PICTURE X(24). 01 QUANT PICTURE S9(9) COMPUTATIONAL. 01 PART PICTURE S9(4) COMPUTATIONAL. EXEC SQL END DECLARE SECTION END-EXEC.</pre>
-------	---

Figure 13 (Part 1 of 2). Examples of Host Variable Declarations

PL/I	<pre>EXEC SQL BEGIN DECLARE SECTION; DCL DESC CHARACTER(24) VARYING; DCL QUANT BINARY FIXED (31); DCL PART BINARY FIXED (15); EXEC SQL END DECLARE SECTION;</pre>
Assembler	<pre>EXEC SQL BEGIN DECLARE SECTION DESC DS H,CL(24) QUANT DS F PART DS H EXEC SQL END DECLARE SECTION</pre>
FORTRAN	<pre>Col. 7 EXEC SQL BEGIN DECLARE SECTION CHARACTER*24 DESC INTEGER QUANT INTEGER*2 PART EXEC SQL END DECLARE SECTION</pre>

Figure 13 (Part 2 of 2). Examples of Host Variable Declarations

The above example also shows the BEGIN and END DECLARE SECTION statements. Observe how the delimiters for SQL statements differ for each language. In all languages, the actual SQL statement is preceded by "EXEC SQL". In COBOL, the end of the command is denoted by "END-EXEC." In PL/I, the usual semicolon (;) is used. There is no trailing delimiter for Assembler or FORTRAN.

The exact rules of placement, continuation, and delimiting of SQL statements are in the host language appendixes. Figure 14 is another example of embedded SQL statements. The INCLUDE SQLCA, WHENEVER, and SELECT statements are shown in each language:

COBOL	<pre> DATA DIVISION. FILE SECTION. WORKING-STORAGE SECTION. EXEC SQL BEGIN DECLARE SECTION END-EXEC. . (host variable declarations) . EXEC SQL END DECLARE SECTION END-EXEC. EXEC SQL INCLUDE SQLCA END-EXEC. PROCEDURE DIVISION. EXEC SQL WHENEVER SQLERROR STOP END-EXEC. EXEC SQL SELECT DESCRIPTION, QONHAND INTO :DESC, :QUANT FROM INVENTORY WHERE PARTNO = :PART END-EXEC. </pre>
PL/I	<pre> EXEC SQL BEGIN DECLARE SECTION; . (host variable declarations) . EXEC SQL END DECLARE SECTION; EXEC SQL INCLUDE SQLCA; EXEC SQL WHENEVER SQLERROR STOP; EXEC SQL SELECT DESCRIPTION, QONHAND INTO :DESC, :QUANT FROM INVENTORY WHERE PARTNO = :PART; </pre>
Assembler	<pre> EXEC SQL BEGIN DECLARE SECTION . (host variable declarations) . EXEC SQL END DECLARE SECTION Col. 72 --- EXEC SQL INCLUDE SQLCA EXEC SQL WHENEVER SQLERROR STOP EXEC SQL SELECT DESCRIPTION, QONHAND * INTO :DESC, :QUANT * FROM INVENTORY * WHERE PARTNO = :PART </pre>

Figure 14 (Part 1 of 2). Examples of Embedded SQL Statements

FORTTRAN	<pre> EXEC SQL BEGIN DECLARE SECTION • (host variable declarations) • EXEC SQL END DECLARE SECTION Col. 6 --EXEC SQL INCLUDE SQLCA EXEC SQL WHENEVER SQLERROR *GO TO 1000 EXEC SQL DECLARE C1 CURSOR FOR * SELECT DESCRIPTION, QONHAND * FROM INVENTORY * WHERE PARTNO = :PART EXEC SQL OPEN C1 EXEC SQL FETCH C1 INTO :DESC, :QUANT EXEC SQL CLOSE C1 • 1000 CALL ERROUT </pre>
-----------------	--

Figure 14 (Part 2 of 2). Examples of Embedded SQL Statements

Note: PL/I, COBOL, and Assembler language programs can also be coded using the “DECLARE-OPEN-FETCH-CLOSE” cursor format required for FORTRAN programs.

Connecting to SQL/DS

In VM/SP environments, the SQL/DS CONNECT statement is not required to establish a connection between SQL/DS and your program. Userid and password checking by VM/SP may be sufficient. SQL/DS does implicit connecting for those environments when an explicit CONNECT is not found.

For explicit CONNECTs, SQL/DS supports the following statement:

```
CONNECT userid IDENTIFIED BY password
```

Both the userid and password must be host variables and must be declared as fixed-length character strings of length 8. For example,

```
CONNECT :USER IDENTIFIED BY :PW
```

“USER” and “PW” are host variables and might contain the following:

```

JONES    <----- USER
JONESPW  <----- PW
12345678 <----- character positions

```

Note that unused positions to the right are padded with blanks.

You must initialize the host variables before the CONNECT statement is executed. To do this, you should code the program to get values for these values via an input file (for example, SYSIN) or via input parameters. When the variables are set from an external source, your program can be executed only by those who know a valid userid and password to provide as input.

CONNECT identifies the user to SQL/DS. In the case where a previously preprocessed program is to be executed, the CONNECT statement in that program identifies the user that is to run the program. This may be the same or a different user than the one that preprocessed it. In either case, the user must have CONNECT authority for the explicit CONNECT, as well as RUN authority for the specific program. The conditions for acquiring authority are discussed in the section, "Authorization" on page 213 in Chapter 2.

Generally speaking, if a CONNECT statement is necessary, it must be the first SQL statement executed in your program. Only SQL *declarative* statements and host language code may precede a CONNECT statement. Figure 15 shows these declarative statements.

```
BEGIN DECLARE SECTION
END DECLARE SECTION
WHENEVER
DECLARE CURSOR
INCLUDE SQLCA
INCLUDE SQLDA
```

Note: SQL declarative statements can also follow the CONNECT statement in some languages.

Figure 15. SQL Declarative Statements

Both the SQLCA structure and the host variable declarations may precede the CONNECT statement.

A CONNECT statement is not required for VM/SP. More information about the implicit connect for VM/SP is contained under "VM/SP Connect Considerations" on page 186.

The CONNECT statement ends the application prolog.

Application Body

The application body is where you place the SQL statements that operate on SQL/DS tables. While there are many SQL statements, most of the day-to-day operations are done using a small subset:

- **SELECT** -- for data retrieval. When more than one row of a table is retrieved, you must use a cursor to retrieve each row. In FORTRAN programs, a cursor must always be used for data retrieval. Cursors are explained under "Retrieving or Inserting Data with a Cursor" on page 19.
- **INSERT** -- to add new rows to an existing table.
- **DELETE** -- to delete rows from a table.
- **UPDATE** -- to change existing rows of a table.

Remember to declare all host variables used in SQL statements, and to properly delimit the statements for the host language.

Application Epilog

The application epilog is the logical end of your SQL/DS application program. To properly end your program:

1. End the current logical unit of work (if one is in progress). **You should always explicitly end your logical units of work.** If you want the changes to be committed, code it explicitly. If you want the changes to be rolled back, code it explicitly.
2. Release your connection to SQL/DS. Others can then use the SQL/DS resources.

CMS Applications

You can issue `COMMIT WORK` or `ROLLBACK WORK` explicitly. The unit of termination is the end of a CMS command or the termination of the user virtual machine. It is at these points where an implicit `COMMIT` or `ROLLBACK WORK` may be invoked.

The implicit `COMMIT` or `ROLLBACK WORK` is automatic for any application that accesses SQL/DS. If an SQL/DS application program is not executed through an `EXEC`, it is considered a “command” and normal, explicit `COMMIT/ROLLBACK WORK` procedures apply. If an SQL/DS application program is executed through an `EXEC`, `COMMIT/ROLLBACK WORK` processing does not occur until the `EXEC` completes.

When implicit `COMMIT` or `ROLLBACK WORK` is invoked at a unit of termination, either a `COMMIT` or a `ROLLBACK` of the logical unit of work occurs, depending upon whether the termination was normal or abnormal.

An application is considered to have terminated normally when it has returned to CMS; or, in single virtual machine mode, when it returns to the SQL/DS calling routine. Any other kind of termination such as `HX`, CMS abend, program check, or any user machine termination is considered an abnormal termination.

In the VM/SP environment, user written interactive SQL applications are provided with an inherent facility to cancel an SQL command without terminating the running application. This cancel facility is invoked via the `SQLHX` immediate command that is established by SQL/DS. The only special processing required of the application is that it be sensitive to the -914 `SQLCODE`.

The terminal operator can cancel long-running SQL commands by entering “`SQLHX`” from the keyboard. This will cause the logical unit of work to be rolled back and an `SQLCODE` of -914 to be returned to the application. If the userid and password were established with an explicit `SQL CONNECT`, it will be necessary to reissue the `CONNECT` or the userid and password will revert to the value established by the implicit `CONNECT`.

The application can modify the basic cancel facility by defining additional names for the SQL/DS-defined SQLHX command or by requesting SQL/DS to remove the SQLHX command and the exit it invokes. These modifications are done using the ARIRCAN macro. For more detail on the ARIRCAN macro interface, see “Recovery Concepts” in the *SQL/Data System Planning and Administration for VM/SP* manual.

For more information on CMS, consult the *Virtual Machine/System Product: CMS User's Guide* or the *Virtual Machine/System Product: CMS Command and Macro Reference* manual.

Summary

Figure 16 summarizes what has been covered so far in this chapter. The pseudo code illustrates a general framework for an SQL/DS application. This framework must, of course, be tailored to suit your own program.

Remember that when used in an SQL statement, host variables must be preceded by a colon. Be sure to declare the host variables used in the CONNECT statement as character strings of fixed length 8.

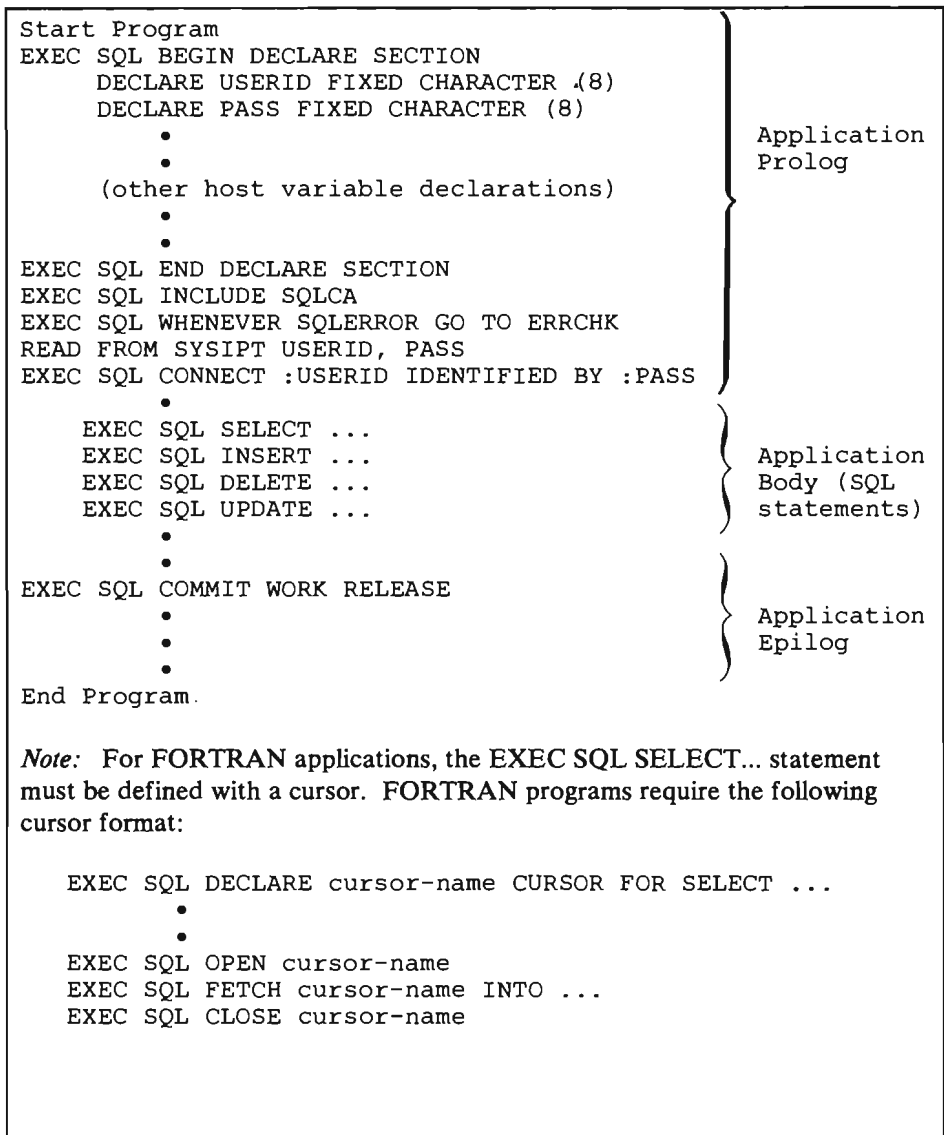


Figure 16. Pseudo Code Framework for Coding Programs

Sample Application Programs

IBM ships five system-dependent sample applications with SQL/DS. The applications are:

ARISAMDB A Data Base Services utility control file. This control file contains commands that create, load, and print sample tables similar to those in the foldout. The control file is shown in the *SQL/Data System Installation for VM/SP* manual.

- ARISASMC** An Assembler language program that manipulates data in the tables created by ARISAMDB and prints results. The source code for the program (ARISASMC) is shown in Appendix E, “Assembler Considerations.”
- ARISCBLC** A COBOL program that manipulates data in the tables created by ARISAMDB and prints results. The source code for the program (ARISCBLC) is shown in Appendix D, “COBOL Considerations.”
- ARISFTN** A FORTRAN program that manipulates data in the tables created by ARISAMDB and prints results. The source code is shown in Appendix F, “FORTRAN Considerations.”
- ARISPLIC** A PL/I program that manipulates data in the tables created by ARISAMDB and prints results. The source code for the program (ARISPLIC) is shown in Appendix C, “PL/I Considerations.”

Each program is an example of using SQL within application programs. You may wish to model your initial programs from the sample applications. IBM supplies EXECs to preprocess, compile, link-edit, and run the sample programs on VM/SP systems.

Each of the following examples assumes that the user (userid) is SQLDBA with a password of SQLDBAPW. If the samples are run under a userid other than SQLDBA, or if SQLDBAPW has been changed, the parameters in the following examples must also be changed. Along with these changes, the host variables used by the CONNECT statement in the sample programs must also be modified to reflect a new userid and/or password.

In operational programs it is generally a better security practice to obtain the userid and password from external parameters, rather than from initialized values of host variables used by CONNECT.

The following is a list of the IBM-supplied EXECs that can be used to preprocess, compile, link-edit, and run the SQL/DS sample programs.

- SQLASMC EXEC Q** The SQL/DS sample Assembler program EXEC.
- SQLCBLC EXEC Q** The SQL/DS sample COBOL program EXEC.
- SQLPLI EXEC Q** The SQL/DS sample PL/I program EXEC.
- SQLFTN EXEC Q** The SQL/DS sample FORTRAN program EXEC.

For example, to preprocess (via the SQL/DS COBOL preprocessor), compile, link edit, and run the SQL/DS sample COBOL program from an SQL/DS user machine, enter the following command:

SQLCBLC

Coding the Program

Contents

More About Search Conditions	99
Additional Types of Constants	99
Double-Byte Character Set (DBCS) Constants	99
Mixing EBCDIC and DBCS Data in Character String Constants	100
Hexadecimal Constants	100
The USER Keyword	100
Nulls	101
Notes on Constructing Search Conditions	103
Rules for Evaluating Predicates	103
Additional Search Predicates	104
BETWEEN Predicate	104
IN Predicate	105
NULL Predicate	105
LIKE Predicate	106
Additions to the SELECT Statement	107
Joining Tables	107
How to Join Tables	107
How SQL/DS Joins Tables	107
A Simple Join Query	108
Joining Another User's Tables	108
Analyzing How a Join Works	109
Nulls Within Join Conditions	110
Joining a Table to Itself	110
Limits on Joins	112
SELECT * As Used in a Join	112
Ordering the Results of a Join	112
Grouping	113
Nulls within Groups	114
Rules for Select-Lists of Grouping Queries	114
Using a WHERE Clause with a GROUP BY Clause	115
The HAVING Clause	116
Combining Joins, WHERE, GROUP BY, HAVING, and ORDER BY ..	117
An Exercise	117
Nesting a Query into Another Query	119
Subqueries That Return a Single Value	122
Subqueries That Return a Null Value	122
Subqueries That Return Many Values	122
Using the IN Predicate with a Subquery	123

Other Subquery Considerations	123
Subqueries That Are Executed Repeatedly: Correlation	124
How to Write a Correlated Subquery	125
How SQL/DS Does Correlation	126
An Exercise	127
Testing for Existence	131
Combining Queries into a Single Query: UNION	132
More About Cursor Management	134
ORDER BY Clause of the DECLARE CURSOR Statement	134
FOR UPDATE Clause of the DECLARE CURSOR Statement	135
More About Data Manipulation	136
Use of Views	140
Creating a View	140
Querying Tables Through a View	142
Modifying Tables Through a View	143
Dropping a View	145
Indicator Variables	146
Dynamically Defined Statements	147
Non-Query Statements	148
Dynamically Defined Queries	151
Parameterized Queries	159
Parameterized Non-Query Statements	163
An Alternative for Parameterized Statements	164
Dynamic Data Conversion	165
The SQL Descriptor Area (SQLDA)	167
PREPARE	172
EXECUTE	175
EXECUTE IMMEDIATE	176
DESCRIBE	177
DECLARE CURSOR Statement for Dynamically Defined Queries	179
OPEN Statement with USING Option	180
FETCH Statement for Dynamically Defined Queries	181
PUT Statement for Dynamically Defined Inserts	181

More About Search Conditions

Additional Types of Constants

There are other types of constants that you can use within expressions, besides the ones discussed in Chapter 1. This section discusses Double-Byte Character Set (DBCS) data, hexadecimal data, and the USER keyword, as used within SQL expressions.

Double-Byte Character Set (DBCS) Constants

Note: If you are not already familiar with the Double-Byte Character Set, and you don't intend to use it, you should skip this section.

Double-Byte Character Set (DBCS) constants can be used in COBOL and PL/I programs, but with two different formats. The SQL/DS preprocessors for COBOL and PL/I also support these constants. DBCS constants are not supported in FORTRAN or Assembler language.

The SQL form of the DBCS constant can be used in dynamic SQL statements and COBOL programs. The SQL form of the DBCS constant is:

```
G'so...si'
```

The shift-out and shift-in characters (“so” and “si”) are single-byte characters, X'0E' and X'0F' respectively. The ellipsis represents any DBCS string. Because they are not within the so/si delimiters, the letter G and the apostrophes (') are single-byte EBCDIC characters, X'C7' and X'7D' respectively. The left byte of a DBCS byte-pair must not be X'0F', since this would signal exit from DBCS encoding. There must be an even number of bytes between the so and the si delimiters. Although character constants require doubling of internal apostrophes to get single apostrophes, no DBCS characters require such doubling.

For PL/I programs, the PL/I form of the DBCS constant must be used for DBCS constants embedded in SQL statements. The DBCS constant for PL/I programs is:

```
so'...'Gsi
```

Unlike the SQL form, the letter G and the apostrophes (') appear inside the so/si delimiters. Therefore they are encoded as DBCS characters. Apostrophe is X'427D' and G is X'42C7'. The so and si are single-byte characters, X'0E' and X'0F' respectively. In the PL/I form of the DBCS constant, DBCS apostrophes (X'427D') must be doubled to obtain a single DBCS apostrophe, similar to the character string constant case for the EBCDIC apostrophe.

The SQL/DS PL/I preprocessor converts PL/I format literals into SQL form constants when they appear in SQL statements. This is done before passing the SQL statement to SQL/DS for processing. Therefore, some SQL/DS messages for incorrect syntax may refer to the SQL form of the constant.

Mixing EBCDIC and DBCS Data in Character String Constants

When the DBCS option is set to YES, a character string constant can contain both EBCDIC and DBCS data. The DBCS strings must be enclosed by so and si. For example:

```
'***so...si***so...si***'
```

where the asterisks (***) represent EBCDIC data and the dots (...) represent DBCS data.

The so...si portions of the data strings must not span across a line. As with unmixed DBCS data, DBCS portions of mixed EBCDIC and DBCS strings do not double the EBCDIC apostrophe (X'7D'). However, X'7D' must be doubled in the EBCDIC portions of the mixed strings.

For more information on mixed EBCDIC and DBCS strings, refer to the "Data Types of Character Strings Constants" section of the *SQL/Data System Planning and Administration for VM/SP* manual.

Hexadecimal Constants

The hexadecimal representation of a constant value must be enclosed within single quote marks, such as:

```
X'2D'      X'C1C2C3C4'      X'4256457D'
```

Each pair of hexadecimal numbers (0-9, A-F) represents a single byte. (Either uppercase or lowercase letters may be used.) Therefore, the number of hexadecimal numbers must be an even number and, when representing a DBCS constant, a multiple of 4 (each DBCS character occupies two bytes in storage).

Hexadecimal constants can be used only to represent character and DBCS data types. The maximum length for hexadecimal constants is 254 hexadecimal numbers; that is, 127 EBCDIC characters or 63 DBCS characters.

Note the following restrictions for hexadecimal constants:

1. They are always treated as VARCHAR data in a SELECT-list.
2. They must not be associated with a host variable in a dynamic statement.
3. They must not be used in an IN predicate.

The USER Keyword

USER is an SQL keyword. It evaluates to the userid of the person who is running the program, regardless of who preprocessed it. That is, USER evaluates to the userid specified in the CONNECT statement. USER behaves exactly like a fixed-length character string constant of length 8, with trailing blanks if the userid has less than eight characters.

This keyword has limited use, however. In particular:

1. You cannot use it in an arithmetic expression (for example, USER+3).
2. You cannot use it in select-lists. (Select-lists are described in the following discussion of the SELECT statement.)
3. You can use it in a predicate where you compare it to a character string (for example, USER = 'JIM').
4. You can, with some restrictions, use it in the SET clause of an UPDATE statement, or in the VALUES clause of an INSERT statement.

Below are valid expressions that incorporate the three data types just discussed:

USER

G' 漢字データ '

X'50C2'

Nulls

SQL/DS allows the existence of nulls in fields of a table. A null is a "non-existent" value; that is, it represents a value that is unknown or not applicable. You can think of a null value as an empty space, or a space reserved for later insertion of data.

When null values occur within expressions, the value of the expression is also null. For example, in this predicate either or both QONORDER and QONHAND may be a null value:

$$\frac{\text{QONHAND} + \text{QONORDER}}{\text{expression1} \quad \text{expression2}} < 100$$

If either QONHAND or QONORDER is null, SQL/DS considers expression1 above as null.

When one of the expressions in a predicate evaluates to null, the *truth-value* of the predicate is unknown. (That is, it is unknown whether a null value is less than 100.) If you combine this predicate with other predicates by using AND, OR, and NOT operators, SQL/DS processes the unknown truth value according to the truth tables in Figure 17. ("?" represents the unknown truth value.)

AND	T	F	?	OR	T	F	?	NOT	T	F	?
T	T	F	?	T	T	T	T	T	F		
F	F	F	F	F	T	F	?	F	T		
?	?	F	?	?	T	?	?	?	?		

Figure 17. Truth Table for Null Values

In any query or data manipulation statement, if the truth-value of the search condition as applied to some row is "unknown," the row does not qualify. (That is, it does not satisfy the search condition and SQL/DS does not select or change it.) For example, suppose that SQL/DS is searching through a table for rows that satisfy the following condition:

PRICE+5.25 < 20.00 AND SUPPNO = 51

Now consider what happens when SQL/DS encounters a row in which the PRICE field is null, but the SUPPNO field is 51:

PRICE+5.25 < 20.00	AND	SUPPNO = 51
"UNKNOWN"	AND	"TRUE"

Because PRICE is null, the expression "PRICE+5.25" is null, thus causing the truth value of the first predicate to be unknown. The SUPPNO field for that particular row is 51, so the second predicate is true. By referring to the truth tables, you can tell whether the row satisfies the search condition:

		TRUE		
		V		
	AND	T	F	?
	T	T	F	?
	F	F	F	F
UNKNOWN	? <---->	?	F	?

"UNKNOWN AND TRUE" evaluate to "UNKNOWN"; the row, therefore, does not satisfy the search condition.

Notes on Constructing Search Conditions

When you are constructing search conditions, there are other considerations you should keep in mind. For example, you should be careful to perform arithmetic only on numeric data types (INTEGER, DECIMAL, SMALLINT, or FLOAT) and to make comparisons only between compatible data types (INTEGER, DECIMAL, SMALLINT, and FLOAT are compatible; all fixed and varying-length character strings are compatible, regardless of length). DBCS data types are only compatible with other DBCS data types. Note also that you can *not* use columns of the type LONG VARCHAR or LONG VARGRAPHIC in your predicates. If you use a host variable in an expression, its host language data type must be compatible with the rest of the expression.

Whenever an arithmetic or comparison operator has operands of two different types, SQL/DS evaluates it in the “greater” of the two types. (FLOAT takes precedence over DECIMAL, DECIMAL takes precedence over INTEGER, and INTEGER takes precedence over SMALLINT.) For example, if the PRICE column is of INTEGER type and has the value 25, the expression PRICE*.5 will evaluate to 12.5, a decimal value. The predicate PRICE*.5=12 is false, because the decimal value forces the predicate to be evaluated in decimal. (Decimal values are stored in System/370 packed decimal format.)

SQL/DS computes all floating point values in normalized form as described in the *System/370 Principles of Operation*, GA22-7000. When a floating point value is stored in a table, it may not be stored exactly as entered. For example, an SQL INSERT statement could specifically insert the constant 3E0 into a field. Internally, however, the value might actually be stored as 2.9999. Floating point values may become even more imprecise when arithmetic operations are performed on them. It is recommended that you use the BETWEEN predicate (described later) when comparing floating point values.

Arithmetic operations between two items of type SMALLINT produce a result of type INTEGER, in order to avoid possible overflow problems (as might easily occur in multiplication). When INTEGER or SMALLINT values are used in a division computation, the result is of type INTEGER, and any remainder is dropped. (See “Data Conversion” on page 76 for more information about data conversion.)

Rules for Evaluating Predicates

SQL/DS observes the following rules when evaluating predicates:

1. When comparing two character strings, SQL/DS uses dictionary ordering. For example:

```
'A' < 'B'  
'A' < 'ABLE'  
'Z' < '35'  
'A1' < 'B'
```

2. When comparing two character strings of fixed length, SQL/DS pads the shorter string on the right with blanks until it equals the length of the longer string. (DBCS strings are padded with X'4040'.) SQL/DS then does the

comparison. For example, if the NAME column of a table is of type CHAR(10), you may write NAME='SMITH' in your search condition, and the condition will be satisfied by the data base value:

'SMITH

3. When comparing two character strings of varying length, SQL/DS performs no padding. To be considered equal, two varying-length strings must have the same length and the same content. For example, 'AA' is *not* equal to 'AA '.
4. In performing an arithmetic operation, if either of the operands is null, the result of the operation is null.
5. No predicates are permitted on data of the type LONG VARCHAR or LONG VARGRAPHIC. Further restrictions on usage of these data types are given in the section "Use of Long Fields" on page 236.
6. When decimal numbers of different scales are compared, the shorter scale is considered extended with trailing zeros sufficient to match the scale of the larger number. For example, 25.45 is equal to 25.4500.
7. When comparing two DBCS character strings, SQL/DS compares the value of the respective data fields in a manner similar to that used for character data types. This single character sequencing is generally of no value for DBCS ordering. Therefore, it is the user's responsibility to specify the sequencing criteria for DBCS data comparisons other than equal or not equal.

Additional Search Predicates

SQL provides four additional types of predicates that you may use in search conditions. These predicates can be used in addition to the standard predicates that compare two expressions. These predicates, described below, are denoted by the keywords BETWEEN, IN, NULL, and LIKE.

You may use the four predicates BETWEEN, IN, NULL, and LIKE alone or with other predicates by using the keywords AND, OR, and NOT to form a search condition.

BETWEEN Predicate

Format:

```
expression1 [NOT] BETWEEN expression2 AND expression3
```

Examples:

```
PRICE BETWEEN 18.00 AND 25.00  
QONHAND + QONORDER BETWEEN :LIMIT1 AND :LIMIT2
```

The three expressions in a BETWEEN predicate are standard expressions constructed from column names, constants, and host variables. The BETWEEN predicate is satisfied if the following condition is true:

```
expression2 <= expression1 <= expression3
```

The BETWEEN predicate is particularly useful in comparing floating point values. The predicate below determines if a value in a column of floating point numbers (called YVALUE) is approximately equal to 3:

```
YVALUE BETWEEN 2.85E0 AND 3.15E0
```

A NOT BETWEEN predicate is true if the corresponding BETWEEN predicate is false.

IN Predicate

Format:

```
expression [NOT] IN (list-of-items)
```

Example:

```
PARTNO IN (221, :P2, :P3, :P4)
```

This predicate enables you to quickly compare the value of an expression with a list of items. The predicate is satisfied if the expression is equal to any of the items in the list (or, if the NOT option is used, not equal to any of the listed items). The items may be constants (for example, 27 or 'BOLT') or host variables (for example, :X). There must be more than one item in the list; separate each item with a comma. A hexadecimal constant cannot be used either as an expression or a list-of-items.

NULL Predicate

Format:

```
column-name IS [NOT] NULL
```

Example:

```
PARTNO IS NULL
```

A row of a table satisfies this predicate if the value of the designated column is (or is not) null. This predicate provides a way for you to explicitly look for null values in tables, or exclude null values from consideration.

Note: You can't use only the NULL keyword in a normal predicate. That is, "WHERE PAY=NULL" is incorrect; but "WHERE PAY IS NULL" is correct.

Similarly, you can't use the NULL keyword in the IN predicate. For example, "WHERE PAY IN (5000,NULL,8000)" is incorrect. You should write "WHERE PAY IN (5000,8000) OR PAY IS NULL."

In addition, you can't use the NULL keyword in a select-list.

LIKE Predicate

Format:

```
column-name [NOT] LIKE {quoted-string | host-variable}
```

The LIKE predicate enables you to search for character string data that *partially* matches a given string.

The column you specify must be of fixed-length or varying-length character or DBCS type. (LONG VARCHAR and LONG VARGRAPHIC are not permitted.) The quoted string or variable on the right side of the LIKE is called a *pattern*. The pattern must have a data type that is compatible with the named column, that is, character to character, DBCS to DBCS, and hexadecimal to either. The pattern may contain any character string, with special meanings for the characters "_" ("___" or X'426D' in DBCS) and "%" ("o/o" or X'426C' in DBCS). The "_" character (or equivalent DBCS value X'426D') represents "any single character." The "%" character (or equivalent DBCS value) represents "any string of zero or more characters." You can use these two special characters within patterns in any combination. The following examples illustrate use of the LIKE predicate:

NAME LIKE '%ANNE%'

(Searches for any name that contains the word ANNE; for example, "LIZANNE," "ANNETTE," or "ANNE.")

NAME LIKE X'426C4F5848F2426C'

(Searches for any occurrence of DBCS character strings containing X'4F5848F2'. Note that the above character string contains the DBCS value X'426C' as the first and last item in the character string. This value is equivalent to the "%" character in the preceding example.)

NAME LIKE G' %汉字% '

This example accomplishes the same as the preceding example, using a DBCS constant.

NAME LIKE ' _A_ '

(Searches for any three-character name that has A as its second letter. To satisfy this pattern, a data value must be of length three, for example, "PAT," "DAN," or "PAM." Its data type may be either fixed-length or varying-length character.)

NAME LIKE :X

(Your program defines a pattern in host variable :X. The pattern may have any combination of "%" and "_" characters. You may change the pattern in the host variable each time the SQL statement containing this predicate is executed. Note that when you use a host variable in a LIKE predicate, the host variable usually should be declared as a varying-length character string.)

A NOT LIKE predicate is true if the corresponding LIKE predicate is false.

Additions to the SELECT Statement

Joining Tables

Joins allow you to write a query against the combined data of two or more tables. (You can also join views. They are discussed under "Use of Views" on page 140.)

How to Join Tables

To join tables, follow these two steps:

1. List in the FROM clause all the tables you wish to join.
2. Specify in the WHERE clause a *join condition*. A join condition expresses some relationship between the tables to be joined.

Note that the data types of the fields involved in the join condition do not have to be identical; they must, however, be compatible. The join condition is evaluated the same as any other search condition, and the same rules for comparisons apply. (These rules are discussed under "Using Expressions as Search Conditions" on page 30.)

How SQL/DS Joins Tables

Conceptually, SQL/DS forms all possible combinations of rows from the indicated tables. For each combination, it tests the join condition. If you don't specify a join condition, SQL/DS returns all possible combinations of rows from tables listed in the FROM clause, even though the rows may be completely unrelated.

A Simple Join Query

The following join query finds the part, description, and price of all parts supplied by supplier 51:

```

DECLARE C1 CURSOR FOR
SELECT INVENTORY.PARTNO, DESCRIPTION, PRICE
FROM QUOTATIONS, INVENTORY
WHERE INVENTORY.PARTNO = QUOTATIONS.PARTNO
AND SUPPNO = 51
    
```

Join
condition.

```

OPEN C1
FETCH C1 INTO :X, :Y:YIND, :Z:ZIND
CLOSE C1
    
```

The WHERE clause above expresses a join condition. If a row from one of the participating tables doesn't satisfy the join condition, that row does not appear in the result of the join. So, if a PARTNO in the INVENTORY table has no matching PARTNO in the QUOTATIONS table (or vice versa), that row does not appear in your result.

Note that more than one table in a join may have a common column name. To identify exactly which column you are referring to, you must use the table name as a prefix, as in the example above. Unique column names don't require a table name prefix.

Here is the query result (based on the example tables shown in the foldout):

PARTNO	DESCRIPTION	PRICE
221	BOLT	.30
231	NUT	.10

Joining Another User's Tables

If you are referring to another user's table, you still must prefix the table name with the userid. Suppose, for example, the tables in the query above belonged to JONES, you would write:

```

DECLARE C1 CURSOR FOR
SELECT JONES.INVENTORY.PARTNO, DESCRIPTION, PRICE
FROM JONES.QUOTATIONS, JONES.INVENTORY
WHERE JONES.INVENTORY.PARTNO = JONES.QUOTATIONS.PARTNO
AND SUPPNO = 51
    
```

_____		_____		_____
_____		_____		_____
_____		_____		_____
_____		_____		_____

OPEN C1
 FETCH C1 INTO :X, :Y:YIND, :Z:ZIND
 CLOSE C1

_____ column name
 _____ table name
 _____ creator

Analyzing How a Join Works

When writing a join query, it is often helpful to mentally go through the query to see what the intermediate results look like.

For example, study the previous SELECT statement. The statement refers to the INVENTORY and QUOTATIONS tables in the foldout. The result of just the join condition looks like this:

SUPPNO	PARTNO	PRICE	DELIVERY_TIME	QONORDER	DESCRIPTION	QONHAND
51	221	.30	10	50	BOLT	650
51	231	.10	10	0	NUT	700
53	222	.25	15	0	BOLT	1250
53	232	.10	15	200	NUT	1100
53	241	.08	15	0	WASHER	6000
54	209	18.00	21	0	CAM	50
54	221	.10	30	150	BOLT	650
54	231	.04	30	200	NUT	700
54	241	.02	30	200	WASHER	6000
57	285	21.00	14	0	WHEEL	350
57	295	8.50	21	24	BELT	85
61	221	.20	21	0	BOLT	650
61	222	.20	21	200	BOLT	1250
61	241	.05	21	0	WASHER	6000
64	207	29.00	14	20	GEAR	75
64	209	19.50	7	7	CAM	50

Each PARTNO in QUOTATIONS was compared to every PARTNO in INVENTORY. When the PARTNO field of both tables matched, a row was formed. The new row contains the combined fields of the “matching” rows. Notice that the only column name that is common to both tables is PARTNO. If the name of the PARTNO column was different in each table, then the PARTNO column of the conceptual result above could have been called either name. This is because of the equality expressed in the join condition. In fact, the select-list could have specified QUOTATIONS.PARTNO instead of INVENTORY.PARTNO, and SQL/DS would have produced identical results.

Now consider what happens when the second part of the WHERE clause (AND SUPPNO=51) is applied:

SUPPNO	PARTNO	PRICE	DELIVERY_TIME	QONORDER	DESCRIPTION	QONHAND
51	221	.30	10	50	BOLT	650
51	231	.10	10	0	NUT	700

The result is further reduced so that only the rows with a supplier number of 51 remain. The entire search condition is now satisfied. Here are the columns that SQL/DS returns based on the select-list:

PARTNO	DESCRIPTION	PRICE
221	BOLT	.30
231	NUT	.10

Nulls Within Join Conditions

Like other predicates, a join condition is never satisfied by a null value. For example, if a row in the INVENTORY table and a row in the QUOTATIONS table both have a null PARTNO, neither row will appear in the result of the join.

Joining a Table to Itself

You can write a query in which you join a table to itself. To join a table to itself, repeat the table name two or more times in the FROM clause. This tells SQL/DS that the join consists of combinations of rows from the same table. When you repeat a table name in the FROM clause, it is no longer unique. Thus, you must give each table name in the FROM clause a unique *join variable* (sometimes called a *table label*).

A join variable can be any string of up to 18 characters beginning with a letter. You use the join variables to resolve column name ambiguities in the select-list and the WHERE clause. For example, the following query finds pairs of quotations for the same part in which the prices differ by more than a factor of two:

```

DECLARE C1 CURSOR FOR
SELECT X.PARTNO, X.SUPPNO, X.PRICE, Y.SUPPNO, Y.PRICE
FROM   QUOTATIONS X, QUOTATIONS Y
WHERE  X.PARTNO = Y.PARTNO
AND    X.PRICE > 2 * Y.PRICE

OPEN C1
FETCH C1 INTO   :PART, :HISUPPNO, :HIPRICE, :LOSUPPNO, :LOPRICE
CLOSE C1

```

If the table is owned by another user, the table name must be qualified in the usual fashion. For example, here is how to write the above query if the creator of the QUOTATIONS table is SCOTT:

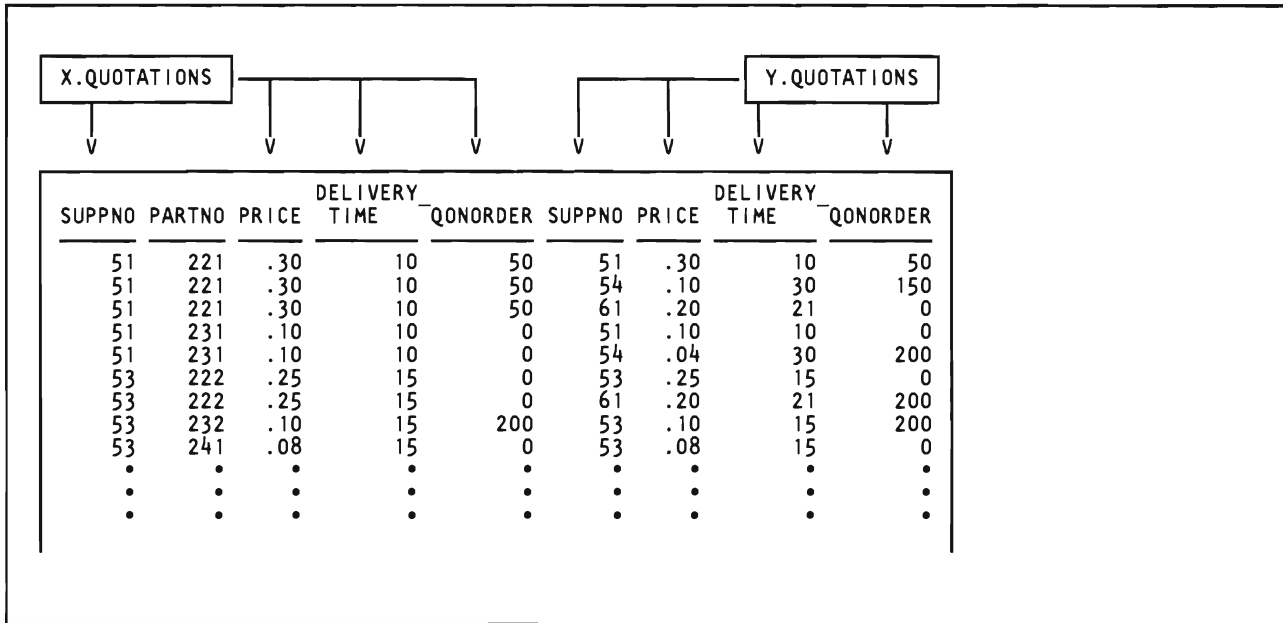
```

DECLARE C1 CURSOR FOR
SELECT X.PARTNO, X.SUPPNO, X.PRICE, Y.SUPPNO, Y.PRICE
FROM   SCOTT.QUOTATIONS X, SCOTT.QUOTATIONS Y
WHERE  X.PARTNO = Y.PARTNO
AND    X PRICE > 2 * Y.PRICE

OPEN C1
FETCH C1 INTO   :PART, :HISUPPNO, :HIPRICE, :LOSUPPNO, :LOPRICE
CLOSE C1

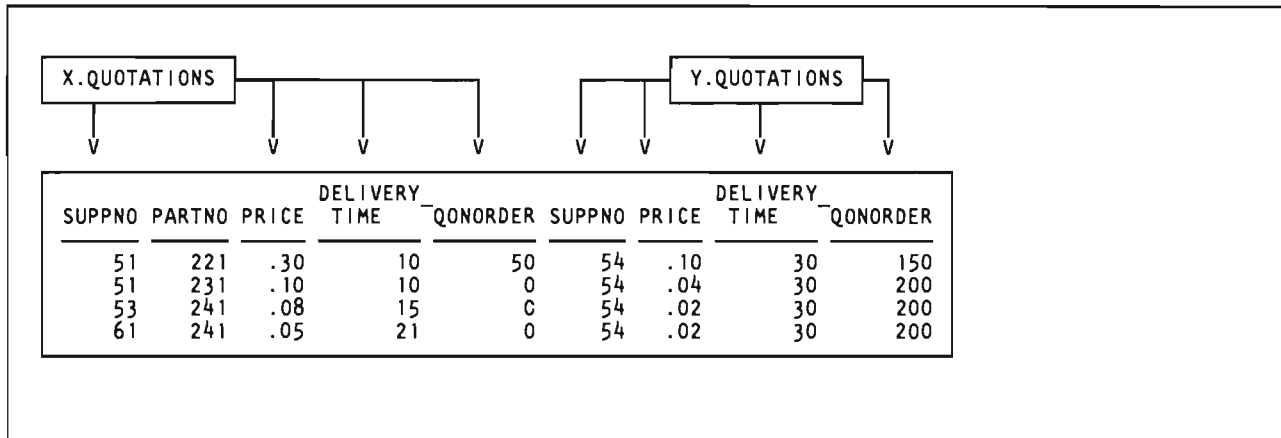
```

This type of join query can also be easily visualized. First, assume there are two tables, X and Y, that are identical to the QUOTATIONS table in the foldout. A partial result of the first join condition (X.PARTNO = Y.PARTNO) looks like this:



This table was formed by taking the PARTNO of the first row of X.QUOTATIONS and comparing it to the PARTNO of the first row of Y.QUOTATIONS. Naturally, they matched (because the X and Y tables are identical), so a row that combined the fields of both was formed. The first row of X was then compared to the second row of Y, and so on, until the end of the Y table was reached. Each time a PARTNO matched, a row was formed in the above table. Every PARTNO of the X table was compared with all the rows of the Y table in a similar fashion, thus completing the first part of the join. (This process is conceptual; you can think of it as nested loops in a normal program.) Note, once again, that PARTNO is the logical meeting point of the tables and could belong to either X.QUOTATIONS or Y.QUOTATIONS.

Now the second join condition (X.PRICE > 2 * Y.PRICE) is applied, producing this result:



Four rows remain in the join table, and from these rows SQL/DS derives your final result via the select-list:

X . PARTNO	X . SUPPNO	X . PRICE	Y . SUPPNO	Y . PRICE
221	51	.30	54	.10
231	51	.10	54	.04
241	53	.08	54	.02
241	61	.05	54	.02

Limits on Joins

Note that the previous example had two join conditions, one relating the two rows by PARTNO, the other by PRICE; a query can have any number of join conditions. Also note that previous examples joined two tables; you can join up to 16 tables.

SELECT * As Used in a Join

The notation "SELECT *" in a join query means "select all the columns of the first table, followed by all the columns of the second table, etc." However, it is not recommended that you use SELECT * for join queries written in programs. It is possible that someone may add a new column to the first table in the join (by an ALTER TABLE statement). If this happens, the columns of the second table are no longer delivered into the correct host variables. By using a normal select-list, however, you avoid this problem.

Ordering the Results of a Join

If a join query uses a qualified column name in its select-list, you can use the same qualified column name in an ORDER BY clause within a cursor definition. For example, SQL/DS accepts ORDER BY X.PARTNO and ORDER BY QUOTATIONS.PARTNO.

Grouping

An earlier section showed how to apply the five built-in functions (SUM, AVG, MIN, MAX, and COUNT) to a table. Previously, however, you could apply the function only to particular fields in rows that satisfied a search condition. For example, the following statement finds the average price of all the parts supplied by supplier number 61:

```
DECLARE C1 CURSOR FOR
SELECT AVG(PRICE)
FROM QUOTATIONS
WHERE SUPPNO = 61

OPEN C1
FETCH C1 INTO :AVG
CLOSE C1
```

The *grouping* feature of SQL/DS permits you to conceptually divide a table into groups of rows with matching values in one or more columns. You can then apply a function to each group. For example, to find the average price of all the parts supplied by *each* supplier:

```
DECLARE C1 CURSOR FOR
SELECT SUPPNO, AVG(PRICE)
FROM QUOTATIONS
GROUP BY SUPPNO

OPEN C1
FETCH C1 INTO :SUPP, :AVGPRICE
CLOSE C1
```

The query yields this result (based on the example tables presented in the foldout):

SUPPNO	AVG(PRICE)
51	.20
53	.14
54	4.54
57	14.75
61	.15
64	24.25

(The DECIMAL value returned by AVG in this example is an approximation. The actual values returned when AVG is used with a column having a DECIMAL data type is discussed in the "Built-In Functions" section of Chapter 1.)

You can group by any column in the table; consider the QUOTATIONS table as grouped by PARTNO:

SUPPNO	PARTNO	PRICE	DELIVERY_TIME	QONORDER
51	221	.30	10	50
54	221	.10	30	150
61	221	.20	21	0
51	231	.10	10	0
54	231	.04	30	200
53	222	.25	15	0
61	222	.20	21	200
.
.
.

←
← (Groups by PARTNO)
←

(Note that the blank space between the groups does not really exist.)

One or more built-in functions can be applied to the groups. The following query finds the maximum, minimum, and average quoted price for each part number group, along with the count of the number of rows in each group (the built-in function COUNT(*) evaluates to the number of rows in the group):

```

DECLARE C1 CURSOR FOR
SELECT PARTNO, MAX(PRICE), MIN(PRICE), AVG(PRICE), COUNT(*)
FROM QUOTATIONS
GROUP BY PARTNO

OPEN C1
FETCH C1 INTO :PART, :HI, :LO, :MID, :NUM
CLOSE C1

```

Nulls within Groups

If any row has a null value in the column you are grouping by (in the previous example, PARTNO), SQL/DS considers each such row as a separate group containing one row.

Rules for Select-Lists of Grouping Queries

When you use the GROUP BY clause in a query, SQL/DS returns only one result row for each group. Therefore, the select-list of such a query can contain only:

- Columns you group by
- Built-in functions on any columns.

For example, this statement is incorrect:

```

DECLARE C1 CURSOR FOR
SELECT SUPPNO, PARTNO, AVG(PRICE)
FROM QUOTATIONS
GROUP BY SUPPNO

OPEN C1
FETCH C1 INTO :SUPPNO, :PARTNO, :AVERAGE
CLOSE C1

```

Wrong!

You cannot include PARTNO in the select-list because PARTNO does not occur in the GROUP BY clause, and is not the operand of some built-in function. Aside from breaking language rules, the above statement is incorrect because a given supplier may supply many parts. It is as though you were asking SQL/DS to return multiple values to the same variable at the same time:

SUPPNO	PARTNO	AVG(PRICE)
51	221	.20
	231	
53	222	.14
	232	
	241	
:	:	:

An impossible result

Using a WHERE Clause with a GROUP BY Clause

A grouping query can have a standard WHERE clause that eliminates non-qualifying rows before the groups are formed and the built-in functions are computed. Write the WHERE clause *before* the GROUP BY clause. The following example query finds the average and minimum price for each part, considering only quotations whose delivery time is less than 30 days:

```

DECLARE C1 CURSOR FOR
SELECT PARTNO, AVG(PRICE), MIN(PRICE)
FROM QUOTATIONS
WHERE DELIVERY_TIME < 30
GROUP BY PARTNO

OPEN C1
FETCH C1 INTO :PART, :A, :B
CLOSE C1

```


The HAVING Clause

You can also apply a qualifying condition to groups, causing SQL/DS to return a result only for those *groups* that satisfy the condition. This is done by the HAVING clause. You can write the HAVING clause *after* the GROUP BY clause. A HAVING clause can contain one or more group-qualifying predicates, connected by ANDs and ORs. Each group-qualifying predicate compares some property of the group, such as MAX(PRICE), with:

1. Another property of the group (HAVING MAX(PRICE) > 2 * MIN(PRICE)); or,
2. A constant (HAVING MAX(PRICE) > 3.00); or,
3. A host variable (HAVING MAX(PRICE) > :LIMIT).

The following example query finds the maximum and minimum prices for various parts in the QUOTATIONS table. The query considers only parts that have at least three quotations and for which the maximum price is more than twice the minimum price:

```
DECLARE C1 CURSOR FOR
SELECT PARTNO, MAX(PRICE), MIN(PRICE)
FROM QUOTATIONS
GROUP BY PARTNO
HAVING COUNT(*) >= 3
AND MAX(PRICE) > 2 * MIN(PRICE)

OPEN C1
FETCH C1 INTO :PART, :HI, :LO
CLOSE C1
```

You can specify DISTINCT as part of the argument of a built-in function in the HAVING clause. Recall that DISTINCT causes SQL/DS to eliminate duplicate values before a function is applied. Thus, COUNT(DISTINCT PARTNO) computes the number of different part numbers. You cannot use DISTINCT in both the select-list and HAVING clause; you can use it only once in a query.

It is possible (though unusual) for a query to have a HAVING clause but no GROUP BY clause. In this case, SQL/DS treats the entire table as one group. Since the table is treated as a single group, it is possible to have, at most, one result row. If the HAVING condition is true for the table as a whole, the selected result (which must consist entirely of built-in functions) is returned; otherwise the “not found” code (SQLCODE = 100) is returned.

Combining Joins, WHERE, GROUP BY, HAVING, and ORDER BY

You can use the various query techniques together in any combination. A query can join two or more tables and can also have a WHERE clause, a GROUP BY clause, a HAVING clause, and, if defined in a cursor, an ORDER BY clause. The precedence of these operations is shown below. Observe that the clauses are applied in the order in which you are to write them:

1. Conceptually, all possible combinations of rows from the listed tables are formed.
2. The WHERE clause, which may contain join conditions, is applied to filter the rows of the conceptual table.
3. The GROUP BY clause is applied to form groups from the surviving rows.
4. The HAVING clause is applied to filter the groups. Only the surviving groups will return a result.
5. The ORDER BY clause determines the order in which the query result is returned.

The actual method used by SQL/DS to arrive at the same result is controlled by the SQL/DS preprocessor.

An Exercise

By now you may be wondering when you need to use which feature. Consider this problem:

Write a query that lists the quantity on hand and minimum quoted price for various parts. Consider only quotations whose delivery time is less than 30 days, and include only parts that have at least two such quotations.

The first thing that must be done is to find in the example tables the names of the columns that contain the requested information so a select-list can be created:

- “quantity on hand” is the QONHAND column of the INVENTORY table.
- “quoted price” is the PRICE column of the QUOTATIONS table, but the problem requests the *minimum* quoted price so the built-in function MIN(PRICE) must be used in the select-list. Notice that the minimum price for a particular part is needed, this means the query will have to group by PARTNO later.
- “various parts” implies PARTNO, but from which table? Observe that the other two items in the select-list are from different tables, so a join is needed, and PARTNO is, obviously, the common field. It must be determined how the PARTNOs are related so a join condition can be written. The problem statement does not express any relationship between the PARTNO fields of the two tables that implies they should be different. Thus, it can be safely assumed that the PARTNOs are related by equality, and that the join condition can be expressed as INVENTORY.PARTNO = QUOTATIONS.PARTNO.

Since the join condition expresses equality, either PARTNO can be used in the select-list. In this example, assume INVENTORY.PARTNO is used to represent the "various parts."

First, the cursor(s) to be used in your program must be defined:

```
DECLARE C1 CURSOR FOR
```

A SELECT clause can now be written:

```
SELECT INVENTORY.PARTNO, QONHAND, MIN(PRICE)
```

The FROM clause must list the two tables used in the join:

```
FROM INVENTORY, QUOTATIONS
```

A WHERE clause is needed because of the join condition:

```
WHERE INVENTORY.PARTNO = QUOTATIONS.PARTNO
```

However, the problem states that for each part only those that have a delivery time of less than 30 days should be considered. This condition needs to be added to the WHERE clause:

```
AND DELIVERY_TIME < 30
```

Note that DELIVERY__TIME is a column in the QUOTATIONS table and is unique among all the column names of the two joined tables, so it does not have to be qualified. So far, the SQL statement is:

```
DECLARE C1 CURSOR FOR
SELECT  INVENTORY.PARTNO, QONHAND, MIN(PRICE)
FROM    INVENTORY, QUOTATIONS
WHERE   INVENTORY.PARTNO = QUOTATIONS.PARTNO
AND     DELIVERY_TIME < 30
```

Next it's necessary to group by PARTNO to find the minimum price for each part, but QONHAND is also in the select-list, so it must be listed in the GROUP BY clause (recall the rules for grouping). Including QONHAND in the GROUP BY clause does not affect the formation of the groups, however, because QONHAND is a property of a given PARTNO. The GROUP BY clause is:

```
GROUP BY INVENTORY.PARTNO, QONHAND
```

Note that you can group by QUOTATIONS.PARTNO if you choose, because of the equality expressed between QUOTATIONS.PARTNO and INVENTORY.PARTNO in the join condition. If you use QUOTATIONS.PARTNO in the GROUP BY clause, however, you must also use it in the select-list:

```
DECLARE C1 CURSOR FOR
SELECT QUOTATIONS.PARTNO, QONHAND, MIN(PRICE)
FROM INVENTORY, QUOTATIONS
WHERE INVENTORY.PARTNO = QUOTATIONS.PARTNO
AND DELIVERY_TIME < 30
GROUP BY QUOTATIONS.PARTNO, QONHAND
```

The problem requests that there be at least two quotations for the part if that part is to be included in the query result; a **HAVING** clause is needed to filter out the unwanted groups:

```
HAVING COUNT(*) >= 2
```

Finally, a nice embellishment is to have SQL/DS return the results in **PARTNO** order.

```
SELECT  INVENTORY.PARTNO, QONHAND, MIN(PRICE)
FROM    INVENTORY, QUOTATIONS
WHERE   INVENTORY.PARTNO = QUOTATIONS.PARTNO
AND     DELIVERY_TIME < 30
GROUP BY INVENTORY.PARTNO, QONHAND
HAVING  COUNT(*) >= 2
ORDER BY 1
```

Now you must position the cursor(s) and identify the corresponding host variables used in your program:

```
OPEN C1
FETCH C1 INTO :PART, :Q, :PRICE
CLOSE C1
```

The complete statement is:

```
DECLARE C1 CURSOR FOR
SELECT  INVENTORY.PARTNO, QONHAND, MIN(PRICE)
FROM    INVENTORY, QUOTATIONS
WHERE   INVENTORY.PARTNO = QUOTATIONS.PARTNO
AND     DELIVERY_TIME < 30
GROUP BY INVENTORY.PARTNO, QONHAND
HAVING  COUNT(*) >= 2
ORDER BY 1

OPEN C1
FETCH C1 INTO :PART, :Q, :PRICE
CLOSE C1
```

Nesting a Query into Another Query

In all previous queries, the **WHERE** clause contained search conditions that SQL/DS used to choose rows for computing expressions in the select-list. SQL/DS also allows a query to refer to a value or set of values computed by another query (called a *subquery*).

Consider this query that finds those quotations for part number 221 in which the price is more than ten cents:

```
DECLARE C1 CURSOR FOR
SELECT SUPPNO, PRICE
FROM QUOTATIONS
WHERE PARTNO = 221
AND PRICE > .10
```

```
OPEN C1
FETCH C1 INTO :S, :P
CLOSE C1
```

Suppose that you want to modify the query so it finds those quotations for part number 221 in which the price is more than twice the minimum quoted price for that part. The problem implies two queries:

1. Find twice the minimum quoted price for part number 221:

```
DECLARE C1 CURSOR FOR
SELECT 2 * MIN(PRICE)
FROM QUOTATIONS
WHERE PARTNO = 221

OPEN C1
FETCH C1 INTO :HIPRICE
CLOSE C1
```

2. Find quotations for part number 221 in which the price is greater than the result of the above query:

```
DECLARE C2 CURSOR FOR
SELECT SUPPNO, PRICE
FROM QUOTATIONS
WHERE PARTNO = 221
AND PRICE > ?

OPEN C2
FETCH C2 INTO :S, :P
CLOSE C2
```

A pseudo code solution for the problem is as follows:

```

EXEC SQL DECLARE CURSOR1 CURSOR FOR
  SELECT SUPPNO, PRICE
  FROM QUOTATIONS
  WHERE PARTNO = 221
  AND PRICE > :HIPRICE
EXEC SQL DECLARE CURSOR2 CURSOR FOR
  SELECT 2 * MIN(PRICE)
  FROM QUOTATIONS
  WHERE PARTNO = 221
EXEC SQL OPEN CURSOR2
EXEC SQL FETCH CURSOR2 INTO :HIPRICE

EXEC SQL OPEN CURSOR1
EXEC SQL FETCH CURSOR1 INTO :S, :P
DO WHILE (SQLCODE=0)
  DISPLAY (S, P)
  EXEC SQL FETCH CURSOR1 INTO :S, :P
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE CURSOR1
EXEC SQL CLOSE CURSOR2

```

← Declare cursor that retrieves quotations.

← Initialize HIPRICE.

← Retrieve quotations.

You can arrive at the same result by using a single query with a subquery. Subqueries must be enclosed in parentheses and may appear in a WHERE clause or a HAVING clause. The result of the subquery is substituted directly into the outer-level predicate in which the subquery appears; thus, there must not be an INTO clause in a subquery. For example, this query solves the above problem:

```

DECLARE C1 CURSOR FOR
SELECT SUPPNO, PRICE
FROM QUOTATIONS
WHERE PARTNO = 221
AND PRICE >
  (SELECT 2 * MIN(PRICE)
   FROM QUOTATIONS
   WHERE PARTNO = 221)
OPEN C1
FETCH C1 INTO :S, :P
CLOSE C1

```

----- Outer-Level Query

----- Subquery

The example subquery above is indented for ease of reading. Remember, however, that the syntax of SQL is fully linear and no syntactic meaning is carried by indentation or by breaking a query into several lines. By using a subquery, the pseudo code is simplified:

```

EXEC SQL DECLARE C1 CURSOR FOR
  SELECT SUPPNO, PRICE
  FROM QUOTATIONS
  WHERE PARTNO = 221
  AND PRICE >
    (SELECT 2 * MIN(PRICE)
     FROM QUOTATIONS
     WHERE PARTNO = 221)
EXEC SQL OPEN C1
EXEC SQL FETCH C1 INTO :S, :P
DO WHILE (SQLCODE=0)
  DISPLAY (S, P)
  EXEC SQL FETCH C1 INTO :S, :P
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE C1

```

← Declare cursor using a subquery that retrieves quotations.

← Retrieve quotations.

The subquery above returned a single value ($2 * \text{MIN}(\text{PRICE})$) to the outer-level query. Subqueries can return either a single value, a null value, or a set of values; each variation has different considerations. In any case, a subquery must have only a single column or expression in its select-list, and must not have an ORDER BY clause.

Subqueries That Return a Single Value

If the subquery returns a single value, as the subquery above did, you can use it on the right side of any predicate in the WHERE clause or HAVING clause. (Exception: Subqueries are not permitted in BETWEEN predicates.)

Subqueries That Return a Null Value

If a subquery returns the null value, the outer-level predicate containing the subquery evaluates to the "unknown" truth-value. How SQL/DS handles "unknown" truth-values is discussed under "Using Expressions as Search Conditions" on page 30.

Subqueries That Return Many Values

If a subquery returns more than one value, you must modify the comparison operators in your predicate ($=$, \neq , $>$, \geq , $<$, \leq) by attaching the suffix ALL or ANY. These suffixes determine how the set of values returned is to be treated in the outer-level predicate. The $>$ comparison operator is used as an example (the remarks below apply to the other operators as well):

expression > (subquery)

denotes that the subquery must return exactly one value (otherwise an error condition results). The predicate is true if the given field is greater than the value returned by the subquery.

expression >ALL (subquery)

denotes that the subquery may return a set of zero, one, or more values. The predicate is true if the given field is greater than each individual value in the returned set. If the subquery returns no values, the predicate is true.

expression >ANY (subquery)

denotes that the subquery may return a set of zero, one, or more values. The predicate is true if the given field is greater than at least one of the values in the set. If the subquery returns no values, the predicate is false.

The following example uses a $>ALL$ comparison to find those quotations having a quoted price greater than all quotations from supplier number 51:

```

DECLARE C1 CURSOR FOR
SELECT  SUPPNO, PARTNO, PRICE
FROM    QUOTATIONS
WHERE   PRICE >ALL
        (SELECT PRICE
         FROM    QUOTATIONS
         WHERE   SUPPNO = 51)

OPEN C1
FETCH C1 INTO :S, :P, :Q
CLOSE C1

```

Using the IN Predicate with a Subquery

Your query can also use the operators **IN** and **NOT IN** when a subquery returns a set of values. For example, the following query lists quotations for those parts having a quantity on hand less than 100:

```

DECLARE C1 CURSOR FOR
SELECT  SUPPNO, PARTNO, PRICE, DELIVERY_TIME
FROM    QUOTATIONS
WHERE   PARTNO IN
        (SELECT PARTNO
         FROM    INVENTORY
         WHERE   QONHAND < 100)

OPEN C1
FETCH C1 INTO :SUPPNO, :PARTNO, :PRICE, :DELIVERY
CLOSE C1

```

The subquery is evaluated once, and the resulting list is substituted directly into the outer-level query. For example, if the subquery above evaluates to part numbers 207, 209, and 295, the outer-level query is evaluated as if its **WHERE** clause were:

```
WHERE PARTNO IN (207,209,295)
```

The list of values returned by the subquery can contain zero, one, or more values. The operator **IN** is equivalent to **=ANY**, and **NOT IN** is equivalent to **≠ALL**.

Other Subquery Considerations

If you link a subquery to an outer query by an unmodified comparison operator such as **=** or **>**, the subquery must not contain a **GROUP BY** or **HAVING** clause. The operator implies that *only one* value will be returned, but a **GROUP BY** clause implies that *more than one* value may be returned. However, a subquery may contain a **GROUP BY** or **HAVING** clause if it is linked by a comparison operator modified by **ALL** or **ANY**, or by a **[NOT] IN** or **[NOT] EXISTS** predicate. (**EXISTS** is described in a following section.)

A subquery may include a join, a grouping, or one or more inner-level subqueries. You may include many subqueries in the same outer-level query, each in its own predicate and enclosed in parentheses

The following example shows how a join and a subquery might be combined to solve a complex problem. The query lists supplier names, addresses, and quoted prices for those parts having a description of 'BOLT'.


```

DECLARE C1 CURSOR FOR
SELECT  NAME, ADDRESS, PARTNO, PRICE
FROM    SUPPLIERS, QUOTATIONS
WHERE   SUPPLIERS.SUPPNO = QUOTATIONS.SUPPNO
AND     PARTNO IN
        (SELECT PARTNO
         FROM    INVENTORY
         WHERE   DESCRIPTION = 'BOLT')

OPEN C1
FETCH C1 INTO :N, :A, :PART, :PRICE
CLOSE C1

```

Subqueries That Are Executed Repeatedly: Correlation

In all the examples of subqueries above, the subquery is evaluated once and the resulting value or set of values is substituted into the outer-level predicate. For example, recall this query from the previous section:

```

DECLARE C1 CURSOR FOR
SELECT SUPPNO, PRICE
FROM QUOTATIONS
WHERE PARTNO = 221
AND PRICE >
      (SELECT 2 * MIN (PRICE)
       FROM QUOTATIONS
       WHERE PARTNO = 221)

OPEN C1
FETCH C1 INTO :S, :P
CLOSE C1

```

The query finds those quotations for part number 221 in which the price is more than twice the minimum quoted price for that part number. Now consider the following problem:

Find those quotations for *every* part number in which the price is more than twice the minimum quoted price for that part number

The subquery needs to be evaluated once for every part number. You can do this by using the *correlation* capability of SQL. Correlation permits you to write a subquery that is executed *repeatedly*, once for each row of the table identified in the outer-level query. This type of “correlated subquery” is used to compute some property of each row of the outer-level table that is needed to evaluate a predicate.

In the first query, the subquery was evaluated *once* for a particular part. In the new problem, the subquery must be evaluated once for *every* part number. One way to solve the problem is to place the query in a cursor definition and open the cursor once for each different part number. The part numbers are determined by using a separate cursor. Here is a pseudo code solution:

```

EXEC SQL DECLARE QUERY1 CURSOR FOR
  SELECT DISTINCT PARTNO
  FROM QUOTATIONS
EXEC SQL DECLARE QUERY2 CURSOR FOR
  SELECT SUPPNO, PRICE
  FROM QUOTATIONS
  WHERE PARTNO = :PARTNO
  AND PRICE >
    (SELECT 2 * MIN(PRICE)
     FROM QUOTATIONS
     WHERE PARTNO = :PARTNO)
EXEC SQL OPEN QUERY1
EXEC SQL FETCH QUERY1 INTO :PARTNO
DO WHILE (SQLCODE = 0)
  EXEC SQL OPEN QUERY2
  EXEC SQL FETCH QUERY2
    INTO :SUPPNO, :PRICE
  DO WHILE (SQLCODE = 0)
    DISPLAY (SUPPNO, PARTNO, PRICE)
    EXEC SQL FETCH QUERY2 INTO :SUPPNO, :PRICE
  END-DO
  EXEC SQL CLOSE QUERY2
  SQLCODE = 0
  EXEC SQL FETCH QUERY1 INTO :PARTNO
END-DO
EXEC SQL CLOSE QUERY1
DISPLAY ('END OF LIST')

```

Retrieve all part numbers listed in QUOTATIONS (eliminate duplicates).

Retrieve SUPPNO and PRICE for parts that are twice the minimum quoted price for that part.

Get a part number.

Evaluate the query for that part.

Get the next part number.

By using a correlated subquery, you can let SQL/DS do the work for you and reduce the amount of code you need to write.

How to Write a Correlated Subquery

To write a query with a correlated subquery, you use the same basic format as an ordinary outer query with a subquery. However, in the FROM clause of the outer query, just after the table name, you place a *correlation variable* (any identifier of up to 18 characters, starting with a letter). The subquery may then contain column references qualified by the correlation variable. For example, if X is a correlation variable, then "X.PARTNO" means "the PARTNO value of the current row of the table in the outer query." The subquery is (conceptually) re-evaluated for each row of the table in the outer query.

The following query solves the problem presented earlier. That is, it finds the quotations for every part number in which the price is more than twice the minimum quoted price for that part number. (Notice that the correlation variable is written in a manner similar to a join variable.)

```

EXEC SQL DECLARE QUERY1 CURSOR FOR
  SELECT SUPPNO, PARTNO, PRICE
  FROM QUOTATIONS X
  WHERE PRICE >
    (SELECT 2 * MIN(PRICE)
     FROM QUOTATIONS
     WHERE PARTNO = X.PARTNO)
EXEC SQL OPEN QUERY1
EXEC SQL FETCH QUERY1
  INTO :S, :P, :PRICE
EXEC SQL CLOSE QUERY1

```

The pseudo code for the correlated subquery solution is:

```
EXEC SQL DECLARE QUERY CURSOR FOR
  SELECT SUPPNO, PARTNO, PRICE
  FROM QUOTATIONS X
  WHERE PRICE >
    (SELECT 2 * MIN(PRICE)
     FROM QUOTATIONS
     WHERE PARTNO = X.PARTNO)
EXEC SQL OPEN QUERY
EXEC SQL FETCH QUERY INTO :SUPPNO, :PARTNO, :PRICE
DO WHILE (SQLCODE=0)
  DISPLAY (SUPPNO, PARTNO, PRICE)
  EXEC SQL FETCH QUERY INTO :SUPPNO, :PARTNO, :PRICE
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE QUERY
```

How SQL/DS Does Correlation

Conceptually, the query is evaluated as follows:

1. QUOTATIONS, the table identified with the correlation variable X, is placed to the side for reference. Let this table be called X, since it is the "correlation table."
2. SQL/DS identifies X.PARTNO with the X table, and uses the values in that column to evaluate the query. (The entire query is evaluated once for every PARTNO in the X table.)

```
EXEC SQL DECLARE QUERY CURSOR FOR
SELECT SUPPNO, PARTNO, PRICE -----> X
FROM QUOTATIONS X _____ |
WHERE PRICE >
  (SELECT 2 * MIN(PRICE)
   FROM QUOTATIONS
   WHERE PARTNO = X.PARTNO) |
-----|-----|-----|
| SUPPNO  PARTNO  PRICE
|-----|-----|-----|
|          221    .30
|          51    .10
|          53    .25
|          .      .
|          .      .
|          .      .
EXEC SQL OPEN QUERY
EXEC SQL FETCH QUERY INTO :S, :P
EXEC SQL CLOSE QUERY
```

Note that PARTNO = X.PARTNO isn't used in the WHERE clause of the outer-level query as it was in the normal subquery; this is because SQL/DS keeps track of which X.PARTNO it is currently evaluating the query for.

Suppose another condition is added to the problem:

Find those quotations for each part number that has a delivery time greater than 20 days, and for which the price is more than twice the minimum quoted price for that part number.

The new query is:

```

EXEC SQL DECLARE QUERY CURSOR FOR
SELECT SUPPNO, PARTNO, PRICE
FROM QUOTATIONS X
WHERE DELIVERY_TIME > 20
AND PRICE >
      (SELECT 2 * MIN(PRICE)
       FROM QUOTATIONS
       WHERE PARTNO = X.PARTNO)
EXEC SQL OPEN QUERY
EXEC SQL FETCH QUERY INTO :S, :P, :PRICE
EXEC SQL CLOSE QUERY

```

The X table in this query is slightly different. Conceptually, whenever there are other conditions besides the one containing the subquery, they are applied to the “correlation table” first. Thus, the X table that is derived from the QUOTATIONS table is:

SUPPNO	PARTNO	PRICE	DELIVERY_TIME	QONORDER
54	209	18.00	21	0
54	221	.10	30	150
54	231	.04	30	200
54	241	.02	30	200
57	295	8.50	21	24
61	221	.10	21	0
61	222	.20	21	200
61	241	.05	21	0

Only rows having a DELIVERY_TIME greater than 20 are included in this "correlation table".

The values 209, 221, 231, 241, 295, and 222 are used for X.PARTNO. Similarly, if you include a GROUP BY clause in the outer-level query, that grouping is applied to the conceptual correlation table first. Thus, if you use a correlated subquery in a HAVING clause, it is evaluated once per *group* of the conceptual table (as defined by the outer-level query’s GROUP BY clause). When you use a correlated subquery in a HAVING clause, the correlated column-reference in the subquery must be a property of each group (that is, must be either the “grouper” column or some other column used with a built-in function).

The use of a built-in function with a correlated reference in a subquery is called a *correlated function*. The argument of a correlated function must be exactly one correlated column (for example, X.PRICE), not an expression. A correlated function may specify the DISTINCT option -- for example, COUNT(DISTINCT X.PARTNO). If so, the DISTINCT counts as the single permitted DISTINCT specification for the outer-level query block (remember that each query-block may use DISTINCT only once).

An Exercise

When would you want to use a correlated subquery? The use of a built-in function is sometimes a clue. Consider this problem:

List quotations whose price is less than the average price *for that part number*.

First you need to determine the select-list items. The problem says to "List quotations." This implies that the query should return at least the number of the supplier making the price quotation, the part number, and the price quotation itself. If you examine the example tables, you'll find that, conveniently enough, all three items (SUPPNO, PARTNO, and QUOTATION) are in the same table (QUOTATIONS). A part of the query can now be constructed:

```
SELECT SUPPNO, PARTNO, PRICE
INTO :SUPPNO, :PARTNO, :PRICE
FROM QUOTATIONS
```

(Assuming only one row is returned). In FORTRAN, a cursor is required.

Next, a search condition (WHERE clause) is needed. The problem statement says, "...whose price is less than the average price for that part number." This means that for each part number in the table, the average price of that part number must be computed. This statement fits exactly the description of a correlated subquery. Some property (average price of the current part number) is being computed for each row. A correlation variable is needed on the QUOTATIONS table:

```
SELECT SUPPNO, PARTNO, PRICE
INTO :SUPPNO, :PARTNO, :PRICE
FROM QUOTATIONS X
```

The subquery needed is simple; it computes the average price for each part number:

```
SELECT AVG(PRICE)
FROM QUOTATIONS
WHERE PARTNO = X.PARTNO
```

This clause tells SQL/DS to compute the subquery once for each PARTNO in the outer-level query table.

The complete SQL statement is:

```
SELECT SUPPNO, PARTNO, PRICE
INTO :SUPPNO, :PARTNO, :PRICE
FROM QUOTATIONS X
WHERE PRICE <
      (SELECT AVG(PRICE)
       FROM QUOTATIONS
       WHERE PARTNO = X.PARTNO)
```

Suppose that instead of listing only the supplier number, part number, and price quoted, that you also list the supplier's name and address. A glance at the example data base will tell you that the information you need (NAME and ADDRESS) is in a separate table (SUPPLIERS). The outer-level query that defines a correlation variable can also be a join query.

When you use joins in an outer-level query, list the tables to be joined in the FROM clause and place the correlation variable next to one of these table names.

To modify the query to list the supplier's name and address, add ADDRESS and NAME to the select-list and change SUPPNO to SUPPLIERS.SUPPNO (to clarify which SUPPNO SQL/DS is to retrieve). The FROM clause must now also include

the SUPPLIERS table, and the WHERE clause must express the appropriate join condition. Here is the modified query:

```
SELECT SUPPLIERS.SUPPNO, NAME, ADDRESS, PARTNO, PRICE
INTO :SUPPNO, :NAME, :ADDRESS, :PARTNO, :PRICE
FROM QUOTATIONS X, SUPPLIERS
WHERE SUPPLIERS.SUPPNO = QUOTATIONS.SUPPNO
AND PRICE <
    (SELECT AVG(PRICE)
     FROM QUOTATIONS
     WHERE PARTNO = X.PARTNO)
```

The above examples show that the correlation variable used in a subquery must be defined in the FROM clause of some query that contains the correlated subquery. However, this containment may involve several levels of nesting. Suppose that the average price of some of the parts may be misleading since some parts only have a few price quotations available. Suppose also that if there are at least three quotations in the data base for a given part, then the average price is a meaningful number to compare a supplier's quotation against. The new statement of the problem is:

List quotations whose price is less than the average price for that part number, but only if there are at least three price quotations for that part in the data base.

The problem implies another subquery, because for each part number in the outer-level query a count of how many exist in the entire QUOTATIONS table is needed:

```
SELECT COUNT(*)
FROM QUOTATIONS
WHERE PARTNO = X.PARTNO
```

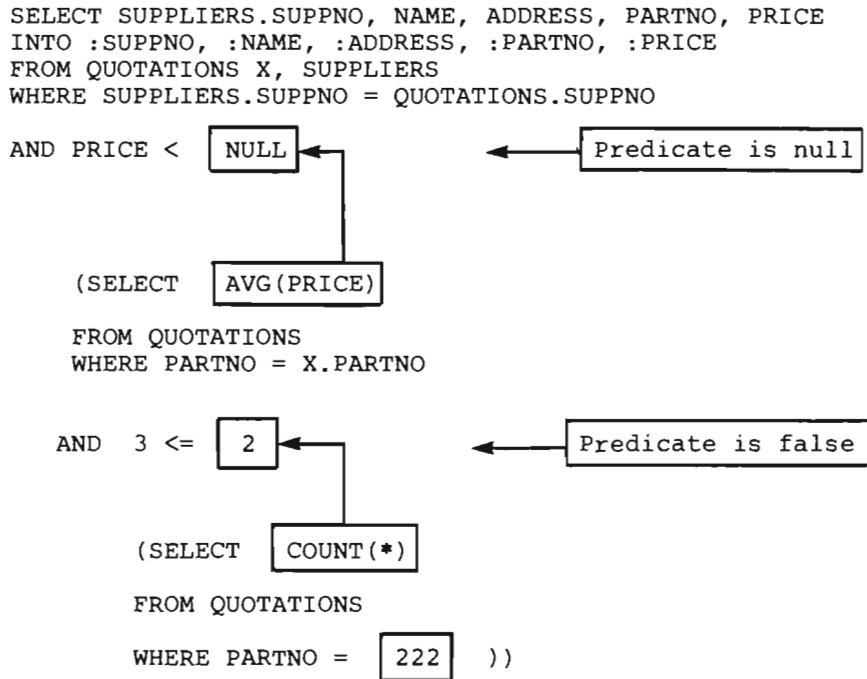
Only if the count is greater than or equal to 3 is an average to be computed:

```
SELECT AVG(PRICE)
FROM QUOTATIONS
WHERE PARTNO = X.PARTNO
AND 3 <=
    (SELECT COUNT(*)
     FROM QUOTATIONS
     WHERE PARTNO = X.PARTNO)
```

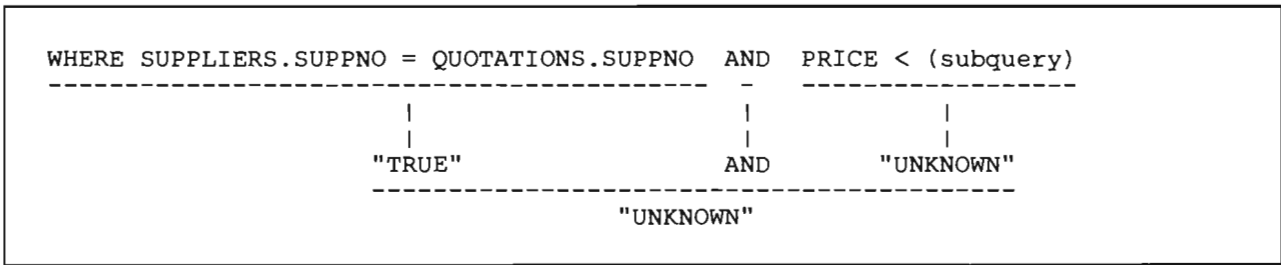
Finally, only those quotations whose price is less than the average price for that part are to be listed:

```
SELECT SUPPLIERS.SUPPNO, NAME, ADDRESS, PARTNO, PRICE
INTO :SUPPNO, :NAME, :ADDRESS, :PARTNO, :PRICE
FROM QUOTATIONS X, SUPPLIERS
WHERE SUPPLIERS.SUPPNO = QUOTATIONS.SUPPNO
AND PRICE <
    (SELECT AVG(PRICE)
     FROM QUOTATIONS
     WHERE PARTNO = X.PARTNO
     AND 3 <=
        (SELECT COUNT(*)
         FROM QUOTATIONS
         WHERE PARTNO = X.PARTNO))
```

If you study the above query, you'll note that it is different from the previous correlated subqueries in that the first subquery may return a null value. Suppose the query is being evaluated for part number 222 and that there are only two quotations for that part in the data base. Working from the bottom to the top, the following occurs:



The inner-most subquery evaluates to 2. Thus, the expression “AND 3 <= 2” is false. Because that expression is false, no rows satisfy the search condition of the next subquery, and no average is computed; a null value is returned to the outer-most query. This causes the predicate “PRICE < (subquery)” to evaluate to the unknown truth value. The join condition “SUPPLIERS.SUPPNO = QUOTATIONS.SUPPNO”, however, is always true:



The following figure is the “AND” truth table for search conditions; “TRUE AND UNKNOWN” causes the search condition in the query to be “UNKNOWN,” as indicated above.

	AND	T	F	?
----->	T	T	F	?
	F	F	F	F
	?	?	F	?

That is, no rows of the data base satisfy the search condition, and no quotation is listed for part number 222 -- exactly the result desired in this case.

Testing for Existence

Format:

```
[NOT] EXISTS (subquery)
```

You can use a subquery to test for the *existence* of a row satisfying some condition. In this case, the subquery is linked to the outer-level query by the predicate EXISTS or NOT EXISTS.

When you link a subquery to an outer query by an EXISTS predicate, the subquery does not return a value. Rather, the EXISTS predicate is true if the answer set of the subquery contains one or more rows, and is false if the answer set of the subquery contains no rows.

The EXISTS predicate is often used with correlated subqueries. The example below lists the suppliers that currently have no entries in the QUOTATIONS table:

```
DECLARE C1 CURSOR FOR
SELECT  SUPPNO, NAME
FROM    SUPPLIERS X
WHERE   NOT EXISTS
        (SELECT      *
         FROM QUOTATIONS
         WHERE SUPPNO = X.SUPPNO)
ORDER BY SUPPNO

OPEN C1
FETCH C1 INTO :S, :N
CLOSE C1
```


You may connect the EXISTS and NOT EXISTS predicates to other predicates by using AND and OR in the WHERE clause of the outer-level query.

Combining Queries into a Single Query: UNION

The UNION operator lets you combine two or more outer-level queries into a single query. Each of the queries connected by UNION is executed to produce an answer set; these answer sets are then combined and duplicate rows are eliminated from the result. If you are using the ORDER BY clause, you must write it after the last query in the UNION. SQL/DS applies the ordering to the combined answer set before it delivers the results to your program via the usual cursor mechanism. None of the queries should have an INTO clause when you are using a cursor.

In COBOL, PL/I, and Assembler language programs, it is possible (though unusual) to write a query using the UNION operator that does not return results via a cursor. In this instance, only one row must be retrieved from the tables and an INTO clause must be placed only in the first query.

The UNION operator is useful when you want to merge lists of values derived from two or more tables. In the following example, the query returns a list of part numbers that are either on order or have a quantity on hand greater than 1000:

```
SELECT PARTNO  
INTO :P:PIND  
FROM QUOTATIONS  
WHERE QONORDER>0
```

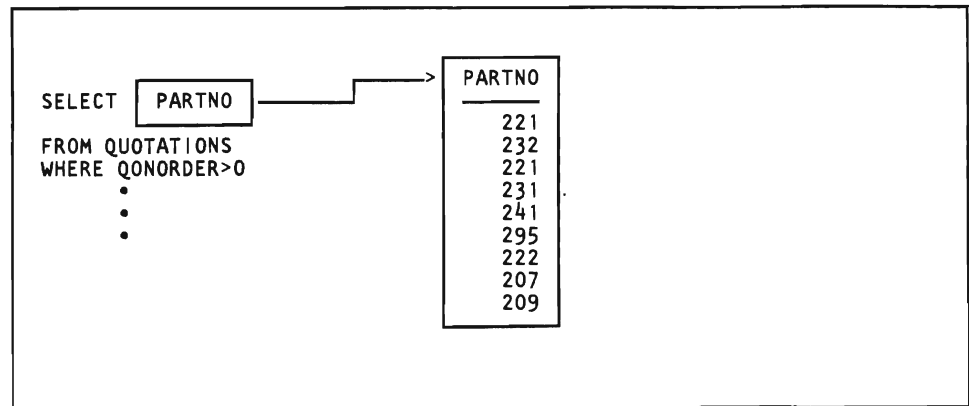
Use INTO only if the query returns one row; otherwise, use a cursor. In FORTRAN programs, a cursor must always be used.

UNION

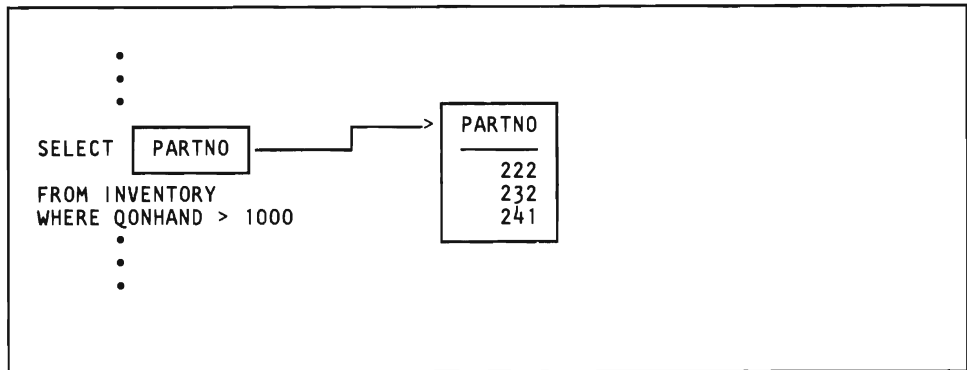
```
SELECT PARTNO  
FROM INVENTORY  
WHERE QONHAND > 1000
```

ORDER BY 1

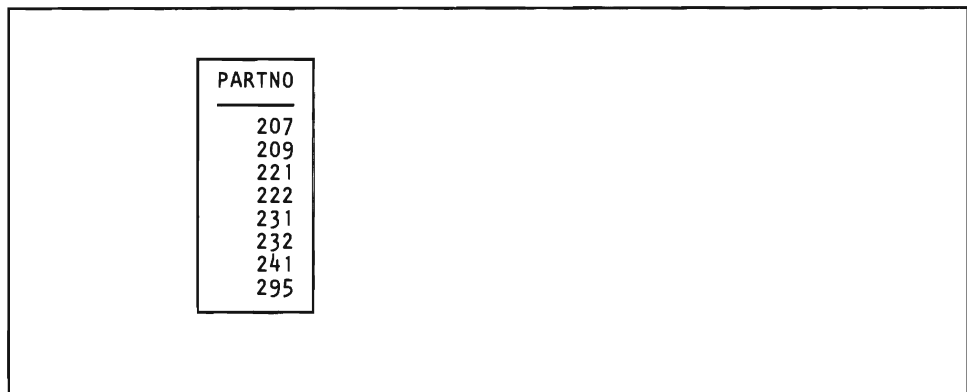
By referring to the example tables in the foldout, it can be seen that the only part not on order is 285. Consequently, the first query returns this answer set:



The second query returns the part numbers having a quantity on hand greater than 1000:



SQL/DS then combines the results of both queries, eliminates the duplicates, and returns the final result in ascending order:



To connect queries by the UNION operator, you must ensure that the queries adhere to the following rules:

1. The data types of corresponding items in the select-lists of all the queries must be identical. For example, if the first item of the select-list of the first query names a column of type INTEGER that permits null values, then the first item of the select-list of *each* query must be an integer column with null values permitted.

SQL/DS strictly enforces the identity of data types for unions: INTEGER is not compatible with SMALLINT, DECIMAL, or FLOAT; character or DBCS columns of different widths are not compatible; and, a column that permits nulls is not compatible with a column that does not permit nulls. However, corresponding items in the select-lists need not have the same name. For example, a query beginning:

```
SELECT X
```

may be in union with a query that begins:

```
SELECT Y
```

provided that X and Y have the same data type.

2. If character constants (literals) are used in the SELECT-clause(s), the constants must be enclosed in single quotes. If the character constant corresponds to another character constant, the shorter constant must be padded with blanks to the length of the longer constant. If the character constant corresponds to a table column, the column must be defined as VARCHAR NOT NULL and the character constant must be padded with blanks to the maximum length defined for that column.
3. If numeric constants are used in the SELECT-clause(s), the constants must be integers. If the numeric constant corresponds to a table column, the column must be defined as INTEGER NOT NULL.
4. An ORDER BY clause, if used, must be placed after the last query in the union. The order-list must contain only integers, not column names. In the example query above, ORDER BY 1 is acceptable but ORDER BY PARTNO is not acceptable.
5. None of the queries in a union may select data of type LONG VARCHAR or LONG VARGRAPHIC.
6. A UNION may not occur inside a subquery.
7. A UNION may not be used in the definition of a view. (Views are discussed in a later section.)

More About Cursor Management

ORDER BY Clause of the DECLARE CURSOR Statement

The ORDER BY clause causes SQL/DS to deliver the rows of the active set in the order specified. You can indicate orderings by specifying an “order specification” (called *o-spec* in the statement syntax). The *o-spec* is a list of column names or integers that refer to select-list items. For example, ORDER BY 3,5 denotes ordering primarily by the third item and secondarily by the fifth item in the select-list. By using integers in the ORDER BY clause, you can order the query result by some selected expression that is not a simple column name. The following query returns results ordered by the expression PRICE*1.10:

```
DECLARE QUERY1 CURSOR FOR
SELECT SUPPNO, PRICE*1.10
FROM QUOTATIONS
WHERE PARTNO = 221
ORDER BY 2
```

You cannot specify ordering by a column that is not in the select-list.

When specifying column names for *o-spec*, the column name must be used within the select-list and must not occur within an expression. For example:

```
DECLARE QUERY2 CURSOR FOR
SELECT PARTNO, PRICE
FROM QUOTATIONS
ORDER BY PARTNO
```

The optional word DESC indicates descending order. ORDER BY 2,5 DESC indicates ascending order on item 2 and descending order on item 5. ASC indicates ascending order, and is the default. Dictionary ordering is used for character-type data. Null values sort last in ascending order; first in descending order. If you do not specify an ORDER BY clause, rows are delivered in an order determined by SQL/DS.

FOR UPDATE Clause of the DECLARE CURSOR Statement

The FOR UPDATE clause tells SQL/DS that you might want to update some columns of the active set. Updating via a cursor is done using the WHERE CURRENT OF clause in an UPDATE statement, which is explained under “Changing Data in a Table: UPDATE” on page 37. You can update only those columns that you list in the FOR UPDATE clause. It is not necessary for a column to appear in the select-list for it to appear in the FOR UPDATE clause. You can update columns that are not explicitly retrieved by the cursor. The FOR UPDATE clause is not required for *deletion* of the current row of a cursor. Deletion via a cursor is done using the WHERE CURRENT OF clause in a DELETE statement, which is explained under “Deleting Data from a Table: DELETE” on page 36.

You *cannot* include both the ORDER BY clause and the FOR UPDATE clause in the same DECLARE CURSOR statement.

Your program may contain many DECLARE CURSOR statements that define different cursors and associate them with different queries. During processing of a program, several of these cursors may be in the open state at one time. The DECLARE CURSOR statement that defines a cursor must occur earlier in the program than any statement operating on that cursor. The DECLARE CURSOR statement does not result in any actual processing when the program is executed (that is, it does not automatically open the cursor).

The “scope” of a cursor-definition is an entire program. Therefore it is an error for two DECLARE CURSOR statements in the same program to use the same cursor-name, even if they are in different blocks or procedures.

You can use DELETE and UPDATE statements to manipulate the data in the current row of the cursor, but only under certain circumstances. The cursor must be open and positioned on a row of the active set before you can attempt a DELETE or UPDATE. For example, a cursor called C1 may be open and positioned on a row of the QUOTATIONS table. When it is in such a state, statements such as the following can be executed:

```
DELETE FROM QUOTATIONS
WHERE CURRENT OF C1

UPDATE QUOTATIONS
SET PRICE = PRICE + :DELTA
WHERE CURRENT OF C1
```

Each such statement deletes or updates exactly one row of the data base: the row that is the current position of cursor C1. If C1 is not correctly positioned on a row of the specified table (for example, if it is not open, or if it is positioned between two rows, or if it is defined on some table other than the one mentioned in the DELETE or UPDATE), the DELETE or UPDATE fails and SQL/DS returns an error code in SQLCODE.

Additional uses of the DECLARE CURSOR statement are discussed under "Dynamically Defined Statements" on page 147.

More About Data Manipulation

Format 2 INSERT:

```
INSERT INTO [creator.]table-name [(list-of-column-names)]
select-statement
```

Example:

```
INSERT INTO MYPARTS
  SELECT PARTNO, DESCRIPTION, PRICE
  FROM SCOTT.PARTS
  WHERE DESCRIPTION = 'PISTON'
```

Authorization:

You can insert data into any table you create. You can insert data into another user's table if you are given the INSERT privilege on that table, or if you have DBA authority. You must have proper SELECT authorization on those tables referenced in the select-statement.

Format 2 of the INSERT statement inserts into an existing table one or more rows. These rows are selected or computed from other tables by a SELECT statement. A SELECT statement used in an INSERT must not have an INTO clause. This is because the destination of the selected items is another table -- not a list of host variables.

When you use a SELECT statement in an INSERT statement, *all* the selected rows are inserted into the target table. SQL/DS does not eliminate duplicate rows before insertion. For example, suppose that you create a table called BOLTS, having columns PARTNO and QONHAND. Suppose also that the new table is presently empty. The following statement inserts into the BOLTS table the relevant values of all rows of the INVENTORY table having a DESCRIPTION of 'BOLT':

```
INSERT INTO BOLTS
  SELECT PARTNO, QONHAND
  FROM INVENTORY WHERE DESCRIPTION = 'BOLT'
```

To eliminate duplicate rows, specify the **DISTINCT** keyword in the **SELECT** statement.

An **INSERT** does not affect any existing rows in the target table or any rows of the table from which the inserted rows were computed (**INVENTORY** in the above example). If the number of columns selected by the **SELECT** statement is not equal to the number of columns needed for the insertion, an error results.

In addition, the columns selected must be type-compatible with the columns into which they are to be inserted. If you insert decimal data into a column of type **INTEGER** or **SMALLINT**, the fractional part of the data is truncated before insertion. If you assign a decimal variable to a decimal column, the number is converted to the precision and scale of the target column. (Extra scale positions are truncated.) A value to be inserted into a column of **CHAR** or **GRAPHIC** data type is padded on the right with blanks (X'40' for **CHAR** data types; X'4040' for **GRAPHIC** data types) to the correct length before insertion. No padding is performed on values inserted into columns of varying length (**VARCHAR** or **VARGRAPHIC**). (SQL/DS conversion rules are summarized under "Data Conversion" on page 76.) You cannot use Format 2 of the **INSERT** statement to insert data of type **LONG VARCHAR** or **LONG VARGRAPHIC**.

Even though SQL/DS does data conversion, you should (if possible) code Format 2 **INSERT** statements so that there is little or no data conversion involved. When SQL/DS does data conversion from source values to target values, it uses more storage internally. It is possible for SQL/DS to exhaust its temporary internal storage when performing operations that involve a large number of data conversions.

The nested **SELECT** statement must not select rows from the same table that is the subject of the **INSERT**, since this might lead to a non-terminating result. If you code such an **INSERT** statement, SQL/DS returns an error.

If SQL/DS detects an error in a Format 2 **INSERT** statement after some rows have been inserted (for example, an attempt to insert a null value into a **NOT NULL** column), SQL/DS stops processing the statement, and returns an error code in the **SQLCA**. If you coded **WHENEVER SQLERROR STOP**, SQL/DS rolls back the current logical unit of work. (The **STOP** condition cannot be used in **FORTRAN** applications.) If you are handling negative return codes via a routine you have coded within the application program (as discussed in the next chapter), the rows that were inserted before the error was detected remain in the table unless you explicitly issue a **ROLLBACK WORK**.

Additional uses of Format 2 of the **INSERT** statement are discussed in the next chapter.

SQL/DS does not impose any logical ordering on the rows of a table; therefore, no facility is provided to specify the "position" in the table of the newly inserted rows. (That is, rows are inserted in SQL/DS-determined order.) You must not use an **ORDER BY** clause in a **SELECT** statement that is associated with a Format 2 **INSERT** statement.

After successful completion of an **INSERT** statement, the variable **SQLERRD(3)** in the return code structure indicates the number of rows that were inserted. If the

returned SQLCODE is non-zero, indicating unsuccessful completion of the statement, the content of SQLERRD(3) is unpredictable.

Format 2 DELETE:

```
DELETE FROM [creator.]table-name WHERE CURRENT OF cursor-name
```

Example:

```
DELETE FROM INVENTORY WHERE CURRENT OF CURSOR2
```

Authorization:

Authorization depends on the table specified in the cursor declaration. You can delete rows of the table named in the cursor declaration if you created that table. If you are not the creator of the table in the cursor declaration, you must be given the DELETE privilege on that table or you must have DBA authority.

Format 2 of the DELETE statement deletes exactly one row of a table. The current position of the cursor determines the row to be deleted. If the cursor name is a reserved keyword, you must use double quotes (") around the cursor name in the DELETE statement. (Notice that the double quotes are *not* used when the cursor is declared.)

The cursor must be open and positioned on a row of the table. In addition, the cursor must meet certain other requirements before you can use it to delete a row as follows:

1. It must be a SELECT statement on one table (not a join).
2. If it contains a subquery, the subquery must not be on the same table as the outer-level query.
3. It must not include DISTINCT or GROUP BY or ORDER BY or UNION or any built-in function such as AVG(PRICE).
4. If you use the BLOCK option on all CREATE PROGRAMs, and you wish to execute a prepared Format 2 DELETE dynamically, the cursor must be a SELECT...FOR UPDATE statement, even if you do not plan to execute any Format 2 UPDATES with the cursor. The FOR UPDATE clause is needed to tell SQL/DS that blocking should be overridden when the SELECT statement is prepared. If you do not use the FOR UPDATE clause in this instance, an error will occur on your DELETE statement.

When the statement is executed, SQL/DS deletes the row indicated by the position of the cursor. The cursor goes into a *between* state in which it remains open but has no current row until you reposition it by a FETCH statement. You cannot use the cursor for further deletions or updates while it is in the between state.

Note that both the COMMIT WORK and ROLLBACK WORK statements automatically close all cursors. A common mistake is to delete a row via a cursor,

commit that change, and then loop backwards to repeat the process. This type of programming construction fails because the first COMMIT WORK closes the cursor.

Format 2 UPDATE:

```
UPDATE [creator.]table-name
SET column-name-1 = expression-1
[, column-name-2 = expression-2] ...
WHERE CURRENT OF cursor-name
```

Example:

```
UPDATE JONES.EMPLOYEE
SET SALARY = 0.00,
    POSITION = 'FIRED'
WHERE CURRENT OF CURSOR1
```

Authorization:

Authorization depends on the table specified in the cursor declaration. You can update rows of the table named in the cursor declaration if you created the named table. If you are not the creator of the table in the cursor declaration, you must be given the UPDATE privilege on those columns you wish to update, or you must have DBA authority.

Format 2 updates exactly one row -- the current row of the indicated cursor. If the cursor name is a reserved keyword, you must use double quotes (") around the cursor name in the UPDATE statement. (Notice that the double quotes are *not* used when the cursor is declared.)

The cursor must be open and positioned on a row of the named table. (Note that both the COMMIT WORK and ROLLBACK WORK statements automatically close all cursors.) The UPDATE statement does not affect the position of the cursor.

The rules for evaluating the SET clause expressions for Format 2 UPDATE statements are identical to those for Format 1. For example, this statement updates the current row of cursor C5. It sets the PRICE field to 2500.00/QONORDER, and then sets the QONORDER field to zero:

```
UPDATE QUOTATIONS
SET PRICE = 2500.00 / QONORDER, QONORDER = 0
WHERE CURRENT OF C5
```

Like Format 1, SQL/DS computes all update values before any updates become effective. Thus, SQL/DS computes the new value of PRICE before setting QONORDER to zero, regardless of the order in which you list the individual updates in the SET clause.

To use an UPDATE statement of Format 2, the named cursor must adhere to these rules:

1. It must be a SELECT statement on one table (not a join).

2. If it contains a subquery, the subquery must not be on the same table as the outer-level query.
3. It must not include `DISTINCT` or `GROUP BY` or `ORDER BY` or `UNION` or any built-in function such as `AVG(PRICE)`.
4. If a particular field of the current row of a cursor is to be updated (for example, `PRICE` in the `UPDATE QUOTATIONS` example), that field must have been included in a `FOR UPDATE` clause in the `DECLARE CURSOR` statement that defined the cursor.

Use of Views

Views allow different users to see different presentations of the same data. For example, several users may be operating on a table of data about employees. The first user may see data about some employees but not others; the second user may see data about all employees but none of their salaries; and the third user may see data about employees joined together with some data from another table. Each of the users in this example is operating on a *view* derived from the real table of data about employees. Each view appears to be a table, and each view has a name of its own.

You can use views with authorization statements to control access to sensitive data. For example, you might use a view based on a `GROUP BY` query to give a user access to the average salary of employees in each department. The view prevents the user from seeing any individual employee salaries.

A view is a dynamic “window” on tables. That is, when you update a real table, you can see the updates through a view. Similarly, when you update a view, SQL/DS updates the real table underlying the view. There are, however, restrictions on modifying tables through a view. “Modifying Tables Through a View” on page 143 covers these restrictions.

Because SQL/DS does not physically store views, you cannot create an index on a view. However, if you create an index on the real table underlying a view, you will improve the performance of queries on the view.

Creating a View

Format:

```
CREATE VIEW [creator.]view-name [(column-name-list)]
AS select-statement
```

Example:

```
CREATE VIEW FASTQUOTES (MFR,PART,DAYS) AS
  SELECT SUPPNO, PARTNO, DELIVERY_TIME
  FROM QUOTATIONS WHERE DELIVERY_TIME < 10
```

Authorization:

You must have the SELECT privilege on the underlying tables to create a view.

The CREATE VIEW statement causes the indicated select-statement to be stored as the definition of a new view. The statement also gives a name to the view, and (optionally) to each column in the view. If you don't specify the column names, the columns of the view inherit the names of the columns from which they are derived.

You must specify new names for the columns of the view if some column of the view is not derived directly from a data field (that is, if a view column is defined as AVG(SALARY) or SALARY+COMMISSION). Columns derived in this manner are often called *virtual columns*. (Virtual columns, naturally, contain *virtual data*.) You must also specify new column names if the selected fields of the view do not have unique names (for example, the view is a join of two tables, each of which has a column named PARTNO).

The data types of the columns of the view are inherited from the columns on which they are defined. If a view column is defined by a built-in function such as AVG(SALARY), the data type of the view column is INTEGER, FLOAT, or DECIMAL. (See "Built-In Functions" on page 31 for a more precise description.)

Here are some other considerations for creating views:

- Internal SQL/DS limitations restrict a view to approximately 140 columns. The number of referenced tables, lengths of column names, and WHERE clauses all further reduce this number.
- If the select-statement in a view definition has a "SELECT *" clause, the view has as many columns as the underlying table. If columns are later added to the underlying table by ALTER statements, the new columns will *not* appear in the view (unless you drop and re-create the view).
- The name of the view must be unique among all the tables, views, and synonyms that you have already created. You can refer to another user's views, if so authorized, by using the person's userid as a prefix (for example, SMITH.FASTQUOTES).
- You can define a view in terms of another view. In other words, the select-statement that defines a view may make reference to one or more other views. In this case, you must observe the limitations listed under "Querying Tables Through a View" on page 142.

- The select-statement in a view definition must not have an ORDER BY clause. Like a table, a view is considered to have no intrinsic ordering. Of course, you can specify an ORDER BY clause when you write queries against the view.
- A select-statement in a view definition cannot contain a UNION operator.
- Host variables are not permitted in a CREATE VIEW statement. (For example, predicates such as PRICE = :X are not permitted.)
- The creator of the view is considered to be the user who preprocessed the program. (Certain exceptions are described under “Dynamically Defined Statements” on page 147.)
- When you define a new view, you receive the same privileges that you had on the underlying table. If you possess these privileges with the GRANT option, you can grant privileges on your view to other users. (See “Granting Privileges to Other Users” on page 62.) If the view is derived from more than one underlying table, you receive only the SELECT privilege, because multi-table views do not permit insertion, deletion, or update. You receive the SELECT privilege on a multi-table view only if you have the SELECT privilege on all the tables from which it is derived. If you have *no* privileges on the underlying table(s), the CREATE VIEW statement returns an error code.
- The special keyword USER, which always evaluates to the userid of the person running the program, can be used in the definition of a view. For example, the following view might be defined on the SQL/DS catalog table SYSCATALOG:

```
CREATE VIEW MYTABLES AS
SELECT * FROM SYSTEM.SYSCATALOG
WHERE CREATOR = USER
```

This view contains only those rows of SYSCATALOG for which the creator is the current user.

The select-statements that define the various views known to the SQL/DS are kept in a catalog called SYSVIEWS. Also, descriptions of views and their columns are kept in SYSCATALOG and SYSCOLUMNS. View names may appear in many other places in the catalogs in place of table names (for example, in SYSTABAUTH). All SQL/DS catalogs are described in the *SQL/Data System Planning and Administration for VM/SP* manual.

Querying Tables Through a View

You can write queries (SELECT statements) against views exactly as if the views were real tables. When you make a query against a view, SQL/DS combines the query with the definition of the view to produce a new query against real stored tables. SQL/DS then processes this query in the usual way. For example, the following query might be written against the view FASTQUOTES that was defined in an example under “CREATE VIEW”:

View Definition for FASTQUOTES:
--

<pre>CREATE VIEW FASTQUOTES (MFR,PART,DAYS) AS SELECT SUPPNO, PARTNO, DELIVERY_TIME * FROM QUOTATIONS WHERE DELIVERY_TIME < 10</pre>

```
SELECT PART,DAYS
FROM FASTQUOTES
WHERE MFR = 51
ORDER BY 2
```

SQL/DS combines this query with the definition of FASTQUOTES and processes the resultant query:

```
SELECT PARTNO, DELIVERY_TIME
FROM QUOTATIONS
WHERE DELIVERY_TIME < 10
AND SUPPNO = 51
ORDER BY 2
```

During the processing of a query on a view, SQL/DS may detect and report errors (via a negative SQLCODE) in either of two phases:

1. The combining of the query with the view-definition (example error: attempt to add together two fields of character-type).
2. The execution of the resulting query on real tables (example error: attempt to fetch a null value when no indicator variable is provided).

You can write almost any kind of query against almost any kind of view. Techniques such as joining, grouping, and nesting can be combined in arbitrary ways, subject to the following limitations:

1. A view column whose definition involves a built-in function cannot be referred to in a WHERE clause, or as the argument of another built-in function in the SELECT clause of a query.
2. A view whose definition involves a GROUP BY cannot be joined with another table or view.
3. A UNION operator cannot be used in the definition of a view.

Modifying Tables Through a View

Like SELECT statements, INSERT, DELETE, and UPDATE statements can be applied to a view just as though it were a real stored table. As described above, the SQL statement that operates on the view is combined with the definition of the view to form a new SQL statement that operates on a stored table. Any data modification made by such a statement is visible to users of the view, or the underlying table, or other views defined on the same table (if the views “overlap” in the modified area).

The following is an example of an update applied to the view FASTQUOTES, showing how the update would be modified by SQL/DS to operate on the real table QUOTATIONS:

View Definition for FASTQUOTES:

```
CREATE VIEW FASTQUOTES (MFR,PART,DAYS) AS
  SELECT SUPPNO, PARTNO, DELIVERY_TIME
  FROM QUOTATIONS WHERE DELIVERY_TIME < 10
```

```
UPDATE FASTQUOTES
SET DAYS = 5
WHERE MFR = 61
AND PART = 241
```

becomes:

```
UPDATE QUOTATIONS
SET DELIVERY_TIME = 5
WHERE SUPPNO = 61
AND PARTNO = 241
AND DELIVERY_TIME < 10
```

You must observe the following limitations when modifying tables through a view:

1. INSERT, DELETE, and UPDATE of the view are not permitted if the view involves any of the following operations: join, GROUP BY, DISTINCT, or any built-in function such as AVG. If one or more of these operations is present in the view definition, the creator of the view does not receive INSERT, DELETE, or UPDATE privileges on the view. Even a user having DBA authority attempting an operation of this type receives a negative SQLCODE.
2. A column of a view can be updated only if it is derived directly from a column of a stored table. Columns defined by expressions such as QONHAND+QONORDER or QONHAND-50 cannot be updated. (These columns are sometimes called virtual columns.) If a view is defined containing one or more such columns, the definer does not receive the UPDATE privilege on these columns. INSERT statements are not permitted on views containing such columns, but DELETE statements are permitted.
3. The ALTER TABLE, CREATE INDEX, and UPDATE STATISTICS statements cannot be applied to a view.

You can use an INSERT statement with a view that does not contain all the columns of the stored table on which it is based. For example, you can insert rows into the view FASTQUOTES even though it does not contain the PRICE and QONORDER columns of the underlying table QUOTATIONS. When such an insert is done, the “invisible” columns receive the null value. If a column that does not permit null values is missing from the view, SQL/DS does not permit insertions to the view.

Note that you can insert or update rows of a view in such a way that they do not satisfy the definition of the view. For example, the view FASTQUOTES is defined by the condition DELIVERY__TIME<10. It is possible to insert rows into

FASTQUOTES having a value greater than 10 in the DAYS field (the field defined on DELIVERY__TIME), or to update a row of FASTQUOTES in such a way that its DAYS value becomes greater than 10. These insertions and updates take effect on the underlying table, QUOTATIONS, but they are not visible in the view FASTQUOTES because the resulting rows do not satisfy the definition of FASTQUOTES. In fact, an update to FASTQUOTES that sets DAYS=12 causes a row to “vanish” from FASTQUOTES (a cursor positioned on the row retains its position, but later scans through FASTQUOTES do not see this row).

Be extremely careful when updating tables through views that may contain duplicate rows. For example, suppose a view PARTS is defined on the INVENTORY table, containing only the columns DESCRIPTION and QONHAND. Since PARTNO is not included in the view, and many parts may have the same description, the user of the view cannot tell which PARTNO corresponds to a given row of the view. If the user positions a cursor on some row where DESCRIPTION = ‘BOLT’, and then updates the current row of this cursor, some row of the stored INVENTORY table is updated. However, since there may be many bolts in the INVENTORY table, and the unique qualifier PARTNO is not part of the view, the user cannot control which bolt is updated. This is not a recommended usage of views.

Dropping a View

Format:

```
DROP VIEW [creator.]view-name
```

Example:

```
DROP VIEW FASTQUOTES
```

Authorization:

You can drop only those views that you have created. You can drop another user’s views only if you have DBA authority.

The DROP VIEW statement drops the definition of the indicated view from the data base. When you drop a view, SQL/DS also:

1. Drops all other views defined in terms of the indicated view. (The underlying tables on which the views are defined are not affected.)
2. Deletes all privileges on the dropped view(s) from the authorization catalogs.
3. Marks invalid all access modules that refer to the dropped views.

The invalid access modules remain in the data base until they are explicitly dropped by a DROP PROGRAM statement. When an invalid access module is next invoked, SQL/DS attempts to regenerate it and restore its validity.

However, if the program contains any SQL statement that refers to a DBSPACE, table, or view that has been dropped, that SQL statement returns an error code at execution time.

If a DROP VIEW statement attempts to drop a view currently in use by another running logical unit of work, SQL/DS queues the DROP VIEW statement until the running logical unit of work terminates.

Indicator Variables

Along with each host variable, you may optionally provide a second variable called an *indicator variable*. Indicator variables can be used to indicate null values on input to SQL/DS (UPDATE and INSERT statements), or output from SQL/DS (INTO-clause of a SELECT statement). These are the rules for using indicator variables:

1. The indicator variable must be of a host language data type equivalent to an SQL/DS SMALLINT.
2. The indicator variable must follow a host variable (called the *main variable*).
3. All indicator variables must be declared in an SQL declare section before they are referred to in SQL statements.
4. Like main variables, you must also precede the indicator variable by a colon (:).
5. If an indicator variable is provided but it is not applicable (for example, if nulls are not allowed for the column), the indicator variable is ignored.

For example:

```
SELECT NAME, ADDRESS  
INTO :NAME:NAMEIND, :ADDR:ADDRIND  
FROM SUPPLIERS WHERE SUPPNO = 51
```

In this example, :NAMEIND serves as the indicator variable for the main variable :NAME, and :ADDRIND serves as the indicator variable for the main variable :ADDR. The value returned in an indicator variable is coded as shown in Figure 18.

Value Returned	Meaning
0	Denotes that the returned value is not null, and has been placed in the associated main variable.

Figure 18 (Part 1 of 2). Values Returned in Indicator Variables

Value Returned	Meaning
<0	Denotes that the returned value is null. The main variable should be ignored.
>0	Denotes that SQL/DS truncated the returned value because the main variable was not of sufficient length. In addition, if the truncated item was a DBCS or character string, the indicator variable contains the length in characters before truncation. The SQLWARN1 warning flag in the SQLCA is set to 'W' whenever a returned character or DBCS string is truncated.

Figure 18 (Part 2 of 2). Values Returned in Indicator Variables

For input (INSERT or UPDATE statements), indicator variables can be used to indicate that a field is to be set to null (when the indicator variable is a negative value). If you provide an indicator variable and assign it a negative value, SQL/DS inserts the null value in the row. A zero or positive value in the indicator causes SQL/DS to insert the value of the main variable. Truncation does not apply to input variables.

Indicator variables are optional. However, if a null value is returned, and you haven't provided an indicator variable, a negative SQLCODE is returned to your program. If your data is truncated and there is no indicator variable, no error condition results (for numeric data). See "Data Conversion" on page 76 for more about truncation.

Do not use indicator variables in search conditions (WHERE clauses). The correct way to test for nulls is via the NULL predicate (described earlier):

WHERE QONHAND IS NOT NULL

Correct

WHERE QONHAND = :SUPP:SIND

Incorrect

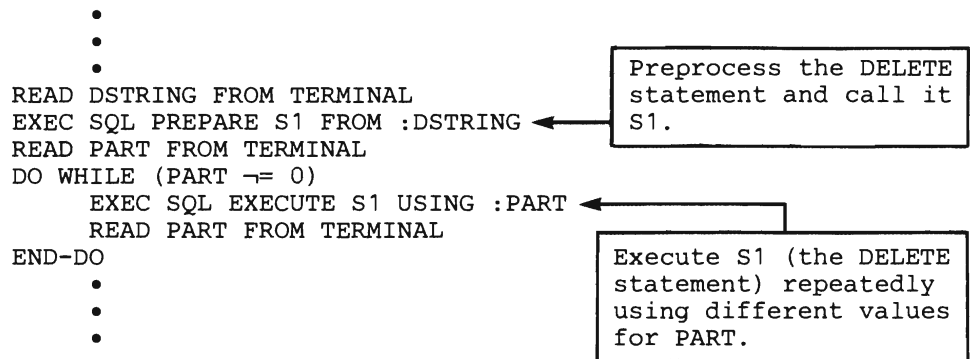
If you use an indicator variable in a WHERE clause, SQL/DS returns a negative SQLCODE to your program.

Dynamically Defined Statements

Note: This topic is more advanced than previous sections. The techniques discussed here are not needed by most application programs. It should also be noted that these dynamically defined statements cannot be used in FORTRAN programs.

Previous sections have described how to code various SQL statements directly into a program and have SQL/DS preprocess them. For some kinds of applications, however, it is desirable to execute SQL statements that are not known until the

preprocessed; the preprocessed statement is also given a name of your choosing. (This name should not be declared as a host variable.) The second step (EXECUTE) causes the statement to be executed using values that you supply for the parameters. Once a statement is prepared, it can be executed many times. Here is the pseudo code:



You should not execute a dynamically defined statement after ending the logical unit of work in which the statement was prepared. If you do, the results are unpredictable.

In routines similar to the above example, the number of parameters and their data types must be known because the host variables that provide input data are declared when the program is being written.

Naturally, this greatly limits the number of different SQL statements that you can read in. In the above example, the only SQL statements that can be executed are those containing a single parameter. This single parameter must be used knowing that it is defined as an integer halfword in the program. For example, the pseudo code above can also process the statements below. (At the terminal, the user types in a statement followed by values for the “?” parameters.)

```
INSERT INTO INVENTORY (PARTNO) VALUES (?)
```

For each value you provide for “?”, the INSERT statement is executed, and a new row is inserted into INVENTORY. The value you enter for “?” is placed in the PARTNO field. The other fields of the table are given the null value.

```
UPDATE INVENTORY SET DESCRIPTION = 'GEAR' WHERE PARTNO = ?
```

For each value you provide for “?”, the UPDATE statement is executed, and the DESCRIPTION column of the INVENTORY table is set to ‘GEAR’.

```
UPDATE INVENTORY SET QONHAND = 0 WHERE PARTNO = ?
```

For each value you provide for “?”, the UPDATE statement is executed, and the QONHAND column of the INVENTORY table is set to 0.

Obviously, there are some applications for this kind of dynamic statement processing, but they are quite specialized. Suppose new parts are added to the inventory. Each part is a different kind of gear, and none of the parts are yet in the warehouse. The input stream for the pseudo code above would be as follows:

```

INSERT INTO INVENTORY (PARTNO) VALUES (?)
301
302
303
304
0
UPDATE INVENTORY SET DESCRIPTION = 'GEAR' WHERE PARTNO = ?
301
302
303
304
0
UPDATE INVENTORY SET QONHAND = 0 WHERE PARTNO = ?
301
302
303
304
0

```

Dynamically Defined Queries

A somewhat more complex facility is needed for executing a dynamically defined SELECT statement. Usually a SELECT statement returns the result of a query into one or more host variables. When the query is read from a terminal at run-time, you cannot know in advance how many and what type of variables to allocate to receive the query result. Therefore, SQL/DS provides a special statement called DESCRIBE, by which a program can obtain a description of the data types of a query result. After using the DESCRIBE statement, the program can dynamically allocate storage areas of the correct size and type to receive the result of the query. If DESCRIBE is used on a prepared SQL statement that was not a SELECT, DESCRIBE returns a special “non-query” indication.

To handle a run-time query, the program first uses the PREPARE statement. As in the previous section, the PREPARE statement preprocesses the SQL statement. The PREPARE step also associates a statement-name with the query. The DESCRIBE statement is then used to obtain a description of the answer set. On the basis of this description, the program dynamically allocates a storage area suitable to hold one row of the result. The program then reads the query result by associating the name of the statement with a cursor and using cursor manipulation statements (OPEN, FETCH, and CLOSE).

The rest of this section describes some techniques for executing dynamically defined queries. The descriptions are not meant to be comprehensive; specific restrictions and statement descriptions are located in a following section.

Dynamically defined queries center around a structure called the SQL Descriptor Area (SQLDA). The SQLDA is usually a based structure; that is, storage for it is allocated dynamically at run time. Figure 19 is a representation of the SQLDA structure with host language independent data type descriptions. Each host language has different considerations for the SQLDA structure. You should read the section on dynamic statements in the appropriate appendix before you attempt to code a program that uses the SQLDA.

```

SQLDA -- a based structure composed of:
  SQLDAID -- character string of length 8
  SQLDABC -- 31-bit binary integer
  SQLN    -- 15-bit binary integer
  SQLD    -- 15-bit binary integer
  SQLVAR  -- an array composed of:
    SQLTYPE -- 15-bit binary integer
    SQLLEN  -- 15-bit binary integer
              SQLPRCSN -- 1-byte (used for DECIMAL)
              SQLSCALE -- 1-byte (used for DECIMAL)
    SQLDATA -- 31-bit binary integer (pointer)
    SQLIND  -- 31-bit binary integer (pointer)
    SQLNAME -- varying-length character string
              of up to 30 characters

```

Figure 19. SQLDA Structure (in Pseudo Code)

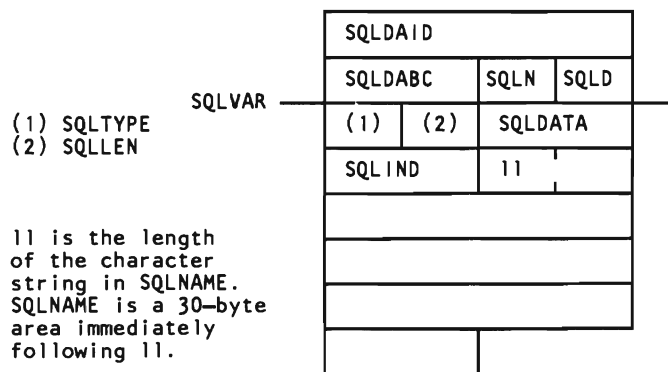
Note that the SQLLEN field is divided into two sub-fields. The sub-fields are used only when working with DECIMAL values. Such usage is described in the following discussion.

To include the descriptor area in your program, specify:

```
INCLUDE SQLDA
```

The INCLUDE SQLDA statement must not be placed in the SQL declare section. As with the SQLCA, you can code this structure directly instead of using the INCLUDE SQLDA statement. If you choose to declare the structure directly, you can specify any name for it. For example, you can call it SPACE1 or DAREA instead of SQLDA.

The following text describes how to process a run-time query. First, you must declare the SQLDA structure. Below is an illustration showing the SQLDA structure as a box; similar illustrations are used in following examples. Remember that SQLDA is a based structure (or, in Assembler, a DSECT); no storage has actually been allocated yet.



The meanings of the various fields are described as they are used. A summary of the meanings of the fields of the SQLDA is presented later for quick reference.

Suppose that a SELECT statement is assigned to a variable called QSTRING. The SELECT statement can be read in from a terminal or SYSIPT, or it can be assigned within the program itself. In this example, the following SELECT statement, which retrieves information from the example tables in the foldout, is read in from the terminal:

```
SELECT DESCRIPTION, QONHAND FROM INVENTORY WHERE PARTNO=221
```

Notice that the SELECT statement has no INTO clause. All SELECT statements that are to be dynamically executed must not have an INTO clause (regardless of whether they return more than one result row).

When the statement is read in, it is assigned to the host variable QSTRING. QSTRING is then preprocessed via the PREPARE statement:

```
READ QSTRING FROM TERMINAL  
EXEC SQL PREPARE S1 FROM :QSTRING
```

Now you can allocate storage for the SQLDA. The techniques for acquiring storage are language dependent. (Refer to the appropriate compiler or Assembler manual.)

Note: The usage of the SQLDA depends on the USING clause option of the DESCRIBE statement. In this section, it is assumed that the NAMES option of the USING clause has been specified. See "DESCRIBE" on page 177 for more detail on the DESCRIBE statement. The amount of storage you need to allocate depends upon how many elements you want to have in the SQLVAR array. Each select-list item must have a corresponding SQLVAR array element. Therefore, the number of select-list items determines how many SQLVAR array elements you should allocate. However, since SELECT statements are specified at run time, it is impossible to know how many select-list items there will be. Consequently, you must guess. Suppose, in this example, that no more than three items are ever expected in the select-list. This means that the SQLVAR array should have a dimension of three, since each item in a select-list must have a corresponding entry in SQLVAR.

Having allocated an SQLDA of what you hope will be adequate size, you must now initialize the SQLDA field called SQLN. SQLN is set to the number of SQLVAR array elements you have allocated. That is, SQLN is the dimension of the SQLVAR array. In this example, you must set SQLN to three. Here's the pseudo code for what was done so far:

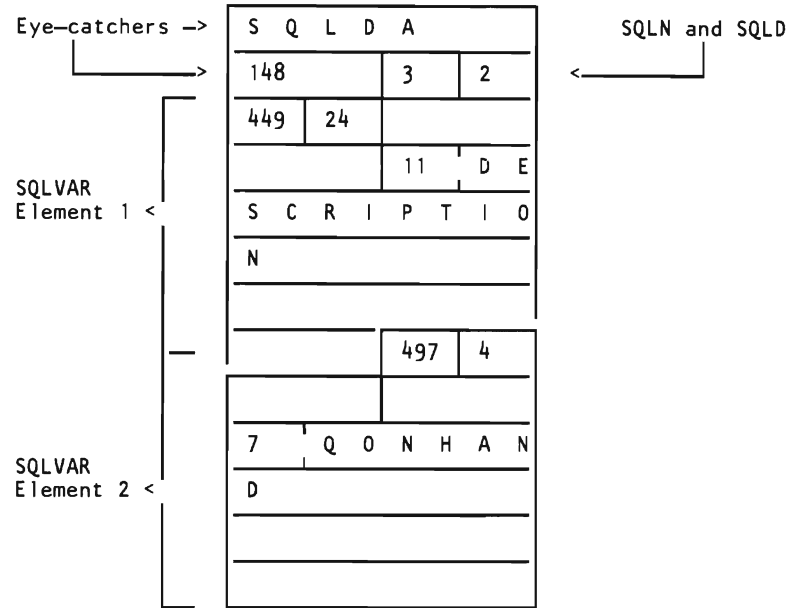
```
Allocate an SQLDA of size 3  
SQLN = 3
```

Having allocated storage, you can now DESCRIBE the statement. (Make sure that SQLN is set before the DESCRIBE.)

```
DESCRIBE S1 INTO SQLDA
```

When the DESCRIBE is executed, SQL/DS places values in the SQLDA for you, these values provide information about the select-list.

The figure below shows the contents of the SQLDA after the DESCRIBE is executed for the example SELECT statement. The third SQLVAR element is not shown because it wasn't used:



SQLDAID and SQLDABC are eye-catcher fields initialized by SQL/DS when a DESCRIBE is executed; you can ignore these for now.

SQLN is not altered by SQL/DS unless you didn't allocate a large enough SQLDA. Suppose, for example, that the SELECT statement contained four select-list expressions instead of two. The SQLDA was allocated with an SQLVAR dimension of three. Naturally, SQL/DS cannot describe the entire select list because there is not enough storage. In this case, SQL/DS sets SQLD to the actual number of select-list expressions; the rest of the structure is ignored. Thus, after a DESCRIBE it is a good practice to check SQLN. If SQLN is less than SQLD, you need to allocate a larger SQLDA based on the value in SQLD:

```
EXEC SQL DESCRIBE S1 INTO SQLDA
IF (SQLN < SQLD)
    Allocate a larger SQLDA using the value of SQLD.
    Reset SQLN to the larger value.
EXEC SQL DESCRIBE S1 INTO SQLDA
END-IF
```

For the example SELECT statement, however, the SQLDA was of adequate size. SQLVAR has a dimension of three, and there are only two select-list expressions. SQLN remains set to 3, and SQL/DS sets SQLD to 2.

If you use DESCRIBE on a non-SELECT statement, SQL/DS sets SQLD to 0. Thus, if your program is designed to process both query and non-query statements, you can describe each statement (after it is prepared) to determine whether it is a query. This example routine is designed to process only query statements, so no test is provided.

Your program must now analyze the elements of SQLVAR. Remember that each element describes a single select-list expression. Consider, again, the SELECT statement that is being processed:

```
SELECT DESCRIPTION, QONHAND FROM INVENTORY WHERE PARTNO=221
```

The first item in the select-list is DESCRIPTION. As illustrated in the beginning of this section, each SQLVAR element contains the fields SQLTYPE, SQLLEN, SQLDATA, SQLIND, and SQLNAME. SQL/DS returns in SQLTYPE a code that describes the data type of the expression and tells whether nulls are applicable. Figure 20 shows how to interpret the codes returned in SQLTYPE:

Data Code	Data Type	Do Nulls Apply?
496	INTEGER	NO
497	INTEGER	YES
500	SMALLINT	NO
501	SMALLINT	YES
484	DECIMAL	NO
485	DECIMAL	YES
480	FLOAT	NO
481	FLOAT	YES
448	VARCHAR	NO
449	VARCHAR	YES
452	CHAR	NO
453	CHAR	YES
456	LONG VARCHAR	NO
457	LONG VARCHAR	YES
468	GRAPHIC	NO
469	GRAPHIC	YES
464	VARGRAPHIC	NO
465	VARGRAPHIC	YES
472	LONG VARGRAPHIC	NO
473	LONG VARGRAPHIC	YES

Figure 20. Data Codes Returned in SQLTYPE

For example, SQL/DS set SQLTYPE to 449 in the first SQLVAR element. This indicates that DESCRIPTION is a VARCHAR column and that nulls are permitted in the column.

SQL/DS sets SQLLEN to the length of the column. For character or DBCS strings, SQLLEN is set to the maximum length in characters of the string. For decimal data, the precision and scale are returned in the first and second bytes, respectively. (Recall that the SQLLEN field has two sub-fields called SQLPRCSN and SQLSCALE for this purpose.) For other data types, SQLLEN is set as follows:

```
SMALLINT -- SQLLEN = 2
INTEGER  -- SQLLEN = 4
FLOAT    -- SQLLEN = 8
```

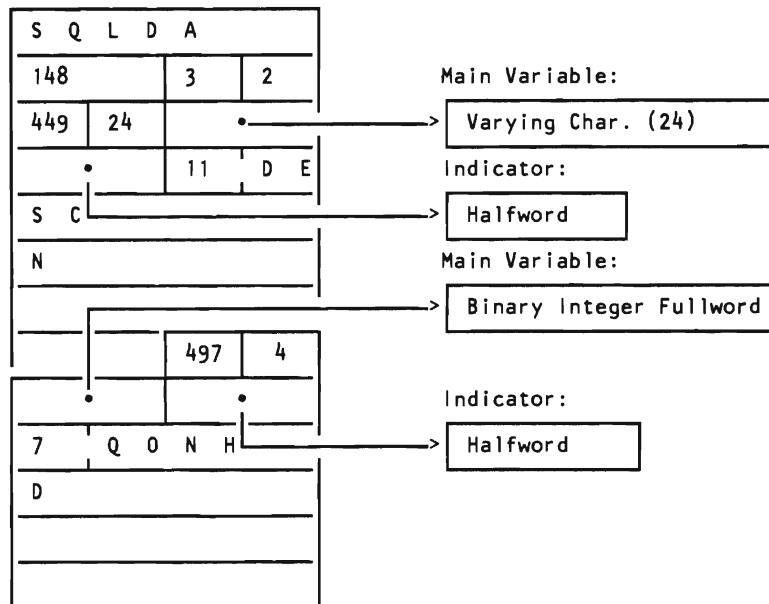
Since the data type of DESCRIPTION is VARCHAR, SQL/DS sets SQLLEN equal to the maximum length of the character string. For DESCRIPTION, that length is 24. Thus, when the SELECT statement is later executed, a storage area large enough to hold a VARCHAR(24) string will be needed. In addition, because nulls are permitted in DESCRIPTION, a storage area for a null indicator variable will also be needed.

The last field in an SQLVAR element is a varying-length character string called SQLNAME. The first two bytes of SQLNAME contain the length of the character data. The character data itself is usually the name of the field used in the select-list expression (DESCRIPTION in the above example). The exceptions to this are select-list items that are unnamed, such as functions (for example, SUM(SALARIES)) and expressions (A+B-C). These exceptions are described in greater detail under "The SQL Descriptor Area (SQLDA)" on page 167.

The second SQLVAR element in the above example contains the information for the QONHAND select-list item. The 497 code in SQLTYPE indicates that QONHAND is an INTEGER column that permits nulls. For an INTEGER data type, SQL/DS sets SQLLEN to 4. SQLNAME contains the character string "QONHAND", and has the length byte set to 7.

After analyzing the result of the DESCRIBE, you can allocate storage for variables that will contain the result of the SELECT statement. For DESCRIPTION, a varying character field of length 24 must be allocated; for QONHAND a binary integer of 31 bits (plus sign) must be allocated. Both QONHAND and DESCRIPTION permit nulls, so you must allocate two additional halfwords to function as indicator variables.

Once the storage is allocated, you must set SQLDATA and SQLIND to point to the appropriate areas. For each element of the SQLVAR array, SQLDATA points to the location where the results are to be placed. SQLIND points to the location where the null indicator is to be placed. Here is what the structure now looks like:



This is the pseudo code for what was done so far:

```

EXEC SQL INCLUDE SQLDA
    .
    .
READ QSTRING FROM TERMINAL
EXEC SQL PREPARE S1 FROM :QSTRING
Allocate an SQLDA of size 3.
SQLN = 3
EXEC SQL DESCRIBE S1 INTO SQLDA
IF (SQLN < SQLD)
    Allocate a larger SQLDA using the value of SQLD.
    Reset SQLN to the larger value.
    EXEC SQL DESCRIBE S1 INTO SQLDA
END-IF
Analyze the results of the DESCRIBE.
Allocate storage to hold select-list results.
Set SQLDATA and SQLIND for each select-list item.

```

Now comes the easy part: retrieving the query result. Dynamically defined queries, as noted earlier, must not have an INTO clause. Thus, all dynamically defined queries must use a cursor. Special forms of the DECLARE, OPEN, and FETCH are used for dynamically defined queries.

The DECLARE CURSOR statement for the example query is as follows:

```

DECLARE C1 CURSOR FOR S1

```

As you can see, the only difference is that the name of the prepared SELECT statement (S1) is used instead of the SELECT statement itself. The dynamic statement *must* be prepared before a cursor is declared for it. (It does not, however, have to be described.)

The actual retrieval of result rows is as follows:


```

EXEC SQL OPEN C1
EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
DO WHILE (SQLCODE = 0)
    DISPLAY (results pointed to by SQLDATA and SQLIND
            for all pertinent SQLVAR elements)
    EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE C1

```

The cursor is opened, and the active set is evaluated. (Note that there are no input host variables needed for the example query. Methods of providing input host variables are discussed later.) The query result rows are then returned using a FETCH. On the FETCH statement there is no list of output host variables. Rather, the FETCH statement tells SQL/DS to return results into the descriptor called SQLDA. The same SQLDA that was set up by DESCRIBE is now being used for the *output* of the SELECT statement. In particular, the results are returned into the storage areas pointed to by the SQLDATA and SQLIND fields of the SQLVAR elements. The meaning of the halfword pointed to by SQLIND is the same as any other indicator variable:

- 0 Denotes that the returned value is not null.
- <0 Denotes that the returned value is null.
- >0 Denotes that the returned value was truncated because the storage area provided was not large enough. If the truncated item was a DBCS or character string, the indicator variable contains the length in characters before truncation.

SQL/DS does not allow you to declare a (non-dynamic) cursor in a program, and then execute dynamically defined statements against it. For example, suppose you code this non-dynamic cursor in your program:

```

DECLARE C1 CURSOR FOR
    SELECT PARTNO, PRICE
    FROM QUOTATIONS
    WHERE SUPPNO = :SUPP
    FOR UPDATE OF PRICE

```

Naturally, you can open C1 and execute statements such as these:

```

UPDATE QUOTATIONS
SET PRICE = PRICE + .10
WHERE CURRENT OF C1

DELETE FROM QUOTATIONS
WHERE CURRENT OF C1

```

However, you cannot read in the above statements at run time and dynamically prepare and execute them. Dynamically defined UPDATE and DELETE statement containing WHERE CURRENT OF clauses will not work when the cursor declaration is non-dynamic.

The next section describes a more general routine in which you can process queries that have parameters in the WHERE clause. It is recommended that you do not

read that section until you have coded some of the simpler dynamic queries discussed thus far.

Parameterized Queries

In the example above, the query that was dynamically executed had no parameters (input host variables) in the WHERE clause:

```
SELECT DESCRIPTION, QONHAND FROM INVENTORY WHERE PARTNO = 221
```

Suppose you wanted to execute the same query a number of times using different values for PARTNO. A parameterized SQL statement (as described under "Non-Query Statements" on page 148) is needed:

```
SELECT DESCRIPTION, QONHAND FROM INVENTORY WHERE PARTNO = ?
```

In previous parameterized SQL statements, the number of parameters and their data types had to be known. What if they are unknown? The DESCRIBE statement, at first glance, is not feasible because it describes only select-lists. With some additional programming, however, you *can* use the DESCRIBE statement to obtain information about the "?" parameters. Specifically, the code must scan the FROM and WHERE clauses to determine which table and column a "?" parameter is associated with. The code can then construct a SELECT statement using those column names in the select-list. For the parameterized statement above, this query can be generated:

```
SELECT PARTNO FROM INVENTORY
```

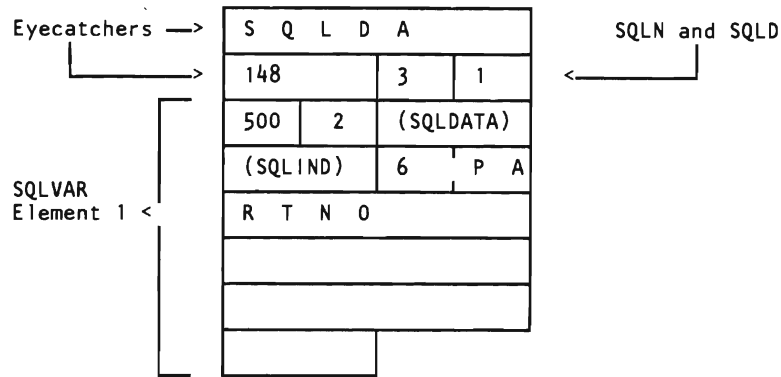
The query (assigned to WSTRING below) can then be preprocessed and described:

```
Allocate an SQLDA of size 3.  
SQLN = 3  
EXEC SQL PREPARE S2 FROM :WSTRING  
EXEC SQL DESCRIBE S2 INTO SQLDA
```

Don't forget to allocate an SQLDA of adequate size and to initialize SQLN. In the example above, it is assumed that no more than three items would appear in the select-list. This means that there can be only three "?" parameters in the WHERE clause since each "?" is equivalent to a select-list expression. (In truth, the scanning routine can easily determine the exact amount of "?" parameters.) Code to allow for re-allocation of a larger structure is appropriate if you believe there may be more than three "?" parameters:

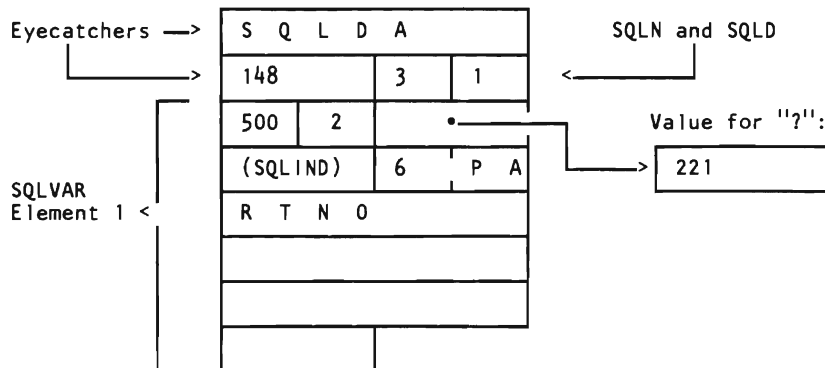
```
EXEC SQL DESCRIBE S2 INTO SQLDA  
IF (SQLN < SQLD)  
    Allocate a larger SQLDA using the value of SQLD.  
    Reset SQLN to the larger value.  
    EXEC SQL DESCRIBE S2 INTO SQLDA  
END-IF
```

Here is what the SQLDA looks like after the fabricated SELECT statement is described. Only the first element of SQLVAR is shown since the others aren't used:



An analysis of the SQLDA shows that there is only one “?” parameter, and that parameter is associated with PARTNO. The SQLTYPE value (500) indicates that PARTNO contains integer halfwords. Thus, you need to allocate a binary integer halfword for the “?” variable. SQLDATA must then be be set to point to this area.

Previously, the SQLDA was used in a FETCH statement and SQL/DS returned query results into the storage areas pointed to by SQLDATA and SQLIND. In other words, the SQLDA was used for *output*. Now, the SQLDA is going to be used to provide *input* values for the WHERE clause via an OPEN statement. When the SQLDA is being used for input, you must assign values to the dynamically allocated storage areas pointed to by SQLDATA. SQLIND is never applicable because you can't use indicator variables in WHERE clauses. In fact, if the SQLTYPE value returned by DESCRIBE shows that the field permits nulls, you should reset SQLTYPE to indicate that nulls are *not* permitted. For example, if the SQLTYPE returned by DESCRIBE is 501, you should set it to 500 before using the SQLDA to provide input. Once the storage for the “?” parameters is allocated you should read in values and assign them to those areas. Here is the completed SQLDA (assuming 221 is read in for “?”):



Once an SQLDA is set up in this fashion, it can be referred to in an OPEN statement that contains a USING clause. For example, a previously declared cursor called C1 is opened using SQLDA:

```
OPEN C1 USING DESCRIPTOR SQLDA
```

Since `SQLDA` currently has 221 in the field pointed to by `SQLDATA`, `C1` is evaluated using that value.

Below is the pseudo code for the complete example. Two `SQLDA`-like structures are used. One is called `SQLDA`, and is the usual structure; the other (declared directly) is called `SQLDA1`. The fields of `SQLDA1` are suffixed with a "1"; for example, `SQLDATA1` and `SQLN1`. An asterisk in position 1 of the pseudo code denotes a comment.

```

EXEC SQL INCLUDE SQLDA
Directly declare SQLDA1.
.
.
* Read in a parameterized query.
*
  READ QSTRING FROM TERMINAL
*
* PREPARE and DESCRIBE the query; set up the output SQLDA.
*
EXEC SQL PREPARE S1 FROM :QSTRING
Allocate an SQLDA of size 3.
SQLN = 3
EXEC SQL DESCRIBE S1 INTO SQLDA
IF (SQLN < SQLD)
  Allocate a larger SQLDA using the value of SQLD.
  Reset SQLN to the larger value.
  EXEC SQL DESCRIBE S1 INTO SQLDA
END-IF
Analyze the results of the DESCRIBE.
Allocate storage to hold select list results.
Set SQLDATA and SQLIND for each select-list item.
*
* Declare a cursor.
*
EXEC SQL DECLARE C1 CURSOR FOR S1
*
* Fabricate a query so PREPARE and DESCRIBE can be used to
* set up the input SQLDA1.
*
Scan the FROM clause and the WHERE clause of QSTRING for "?"
parameters and generate an appropriate query in WSTRING.
Allocate an SQLDA1 of size 1 (1 was obtained from the scan).
SQLN1 = 1
EXEC SQL PREPARE S2 FROM :WSTRING
EXEC SQL DESCRIBE S2 INTO SQLDA1
Analyze the results of the DESCRIBE.
Reset SQLTYPE to reflect that there is no indicator variable.
Allocate storage to hold the input values (the "?" values).
Set SQLDATA1 for each "?" parameter value.
*
* Read in input parameters and retrieve the query results via
* cursor C1. Note that the pseudo code reads in only one "?"
* parameter. Your actual code must provide for the possibility
* that more than one "?" parameter might be provided.
*
  READ PARM FROM TERMINAL
  DO WHILE (PARM ^= 0)
    Assign PARM to area pointed to by SQLDATA1.
    EXEC SQL OPEN C1 USING DESCRIPTOR SQLDA1
    EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
    DO WHILE (SQLCODE = 0)
      DISPLAY (results pointed to by SQLDATA and SQLIND)
      EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
    END-DO
    EXEC SQL CLOSE C1
    DISPLAY ('ENTER ANOTHER VALUE OR 0')
    READ PARM FROM TERMINAL
  END-DO
  DISPLAY ('END OF QUERY')

```

Of course, the pseudo code above must be modified to suit your own purposes.

Parameterized Non-Query Statements

In "Non-Query Statements" on page 148, parameterized statements were introduced; it was necessary, however, to know the number of "?" parameters and their data types before run-time. In the preceding section it was shown how you might analyze a parameterized query so that a SELECT statement could be generated and subsequently described.

The same principle can be used for parameterized non-query statements. For example, suppose this DELETE statement is read from the terminal and assigned to DSTRING:

```
DELETE FROM QUOTATIONS WHERE PARTNO = ? AND SUPPNO = ?
```

Suppose also that the amount of "?" parameters and their corresponding data types are unknown before run time. The same routine that you coded to scan the FROM and WHERE clauses of SELECT statements can be used to scan the above DELETE statement. Then, a SELECT statement containing the relevant columns can be constructed:

```
SELECT PARTNO, SUPPNO FROM QUOTATIONS
```

This SELECT statement is then prepared and described as in the previous section. The setup of the SQLDA is also identical: once the SQLDA is analyzed, space to hold the "?" values is allocated, and the "?" values are read in and assigned to these locations. Once again, the SQLDA will be used for *input* to the WHERE clause of the SQL statement; no indicator variables are allowed. Because the statement is a non-query statement, the SQLDA is pointed to in the EXECUTE statement. (There is no OPEN for non-queries.) Here is the pseudo code for a parameterized non-query statement.

```
EXEC SQL INCLUDE SQLDA
      •
      •
READ DSTRING FROM TERMINAL
Scan the FROM clause and the WHERE clause of DSTRING for "?"
  parameters and generate an appropriate query in WSTRING.
Allocate an SQLDA of size 2 (2 was obtained from the scan).
SQLN = 2
EXEC SQL PREPARE S2 FROM :WSTRING
EXEC SQL DESCRIBE S2 INTO SQLDA
Analyze the results of the DESCRIBE.
Reset SQLTYPE to reflect that there is no indicator variable.
Allocate storage to hold the input values (the "?" values).
Set SQLDATA for each "?" parameter value.

EXEC SQL PREPARE S1 FROM :DSTRING
Read "?" parameter values from the terminal.
* A zero parameter value terminates the DO loop.
DO WHILE (parameters ≠ 0)
  Assign the values to the storage allocated for
  input variables.
  EXEC SQL EXECUTE S1 USING DESCRIPTOR SQLDA
  Prompt user for more values.
  Read "?" parameter values from the terminal.
END-DO
      •
      •
```

Note that you may need a more complex scanning routine depending on how many different non-query statements you wish to process. For example, the above routine would have to be modified if you wanted to process INSERT statements. In that case, you would have to scan the INTO clause. Note also that indicator variables are permitted when providing input to the INSERT statement via EXECUTE. This is because a normal (not dynamically defined) INSERT statement also permits indicators. If indicator variables are permitted in normal usage, they are permitted in the dynamically defined case.

An Alternative for Parameterized Statements

Previous sections on parameterized statements (both query and non-query) relied on a scanning routine that generated a query. Once this query was generated, DESCRIBE was used to obtain information about the columns and expressions associated with a “?” parameter.

If you have not coded a scanning routine that generates a query, there is a simple alternative: have the user describe the “?” parameters for you, and fill in the SQLDA yourself. There is no rule that says you must use a DESCRIBE to fill in the SQLDA. When using the SQLDA for input or output, SQL/DS doesn't care who filled in the SQLDA, as long as the needed values are there.

When you use the SQLDA for input (which is always the case for “?” parameters), not all fields have to be filled in. Specifically, SQLDAID, SQLDABC, and the SQLVAR field called SQLNAME need not be filled in. Thus, if you choose this method, you will need to ask the user for the following:

1. How many “?” parameters are there?
2. What are the data types of these parameters (and lengths, if character)?
3. Do you want an indicator variable?

In addition, if the routine is to handle both query and non-query statements, you may want to ask the user what category of statement it is. (Alternatively, you can write code to look for the SELECT keyword.)

The code that interrogates the user and sets up the SQLDA would take the place of the scanning routine and DESCRIBE in the previous sections:

With a Scanning Routine:

```
•  
•  
READ DSTRING FROM TERMINAL  
Scan the FROM and WHERE clauses of DSTRING for "?"  
parameters and generate an appropriate query in WSTRING.  
Allocate an SQLDA of size 2 (2 was obtained from the scan).  
SQLN = 2  
EXEC SQL PREPARE S2 FROM :WSTRING  
EXEC SQL DESCRIBE S2 INTO SQLDA  
Analyze the results of the DESCRIBE.  
Reset SQLTYPE to reflect that there is no indicator variable.  
Allocate storage to hold the input values (the "?" values).  
Set SQLDATA for each "?" parameter value.  
•  
•
```

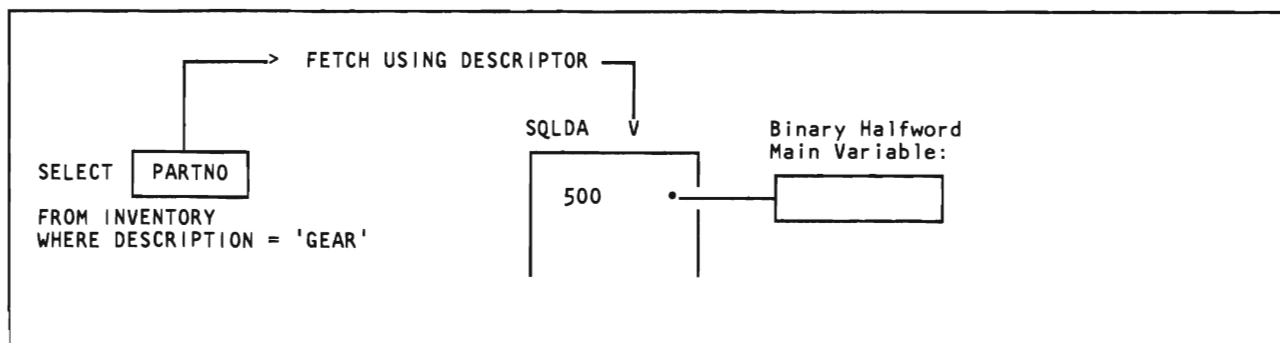
Without a Scanning Routine:

```
•  
•  
READ DSTRING FROM TERMINAL  
Interrogate user for number of "?" parameters.  
Allocate an SQLDA of that size.  
Set SQLN and SQLD to the number of "?" parameters.  
For each "?" parameter:  
Interrogate user for data types, lengths, and  
indicators.  
Set SQLTYPE and SQLLEN.  
Allocate storage to hold the input values  
(the "?" values).  
Set SQLDATA and SQLIND (if applicable) for each  
"?" parameter.  
•  
•
```

The statement can then be processed in the usual manner.

Dynamic Data Conversion

In previous uses of the SQLDA for input or output, SQLTYPE *always* described the data type of the storage area pointed to by SQLDATA. In the following example, the type code 500 (originally obtained via a DESCRIBE of the SELECT statement) describes the data type of the main variable.

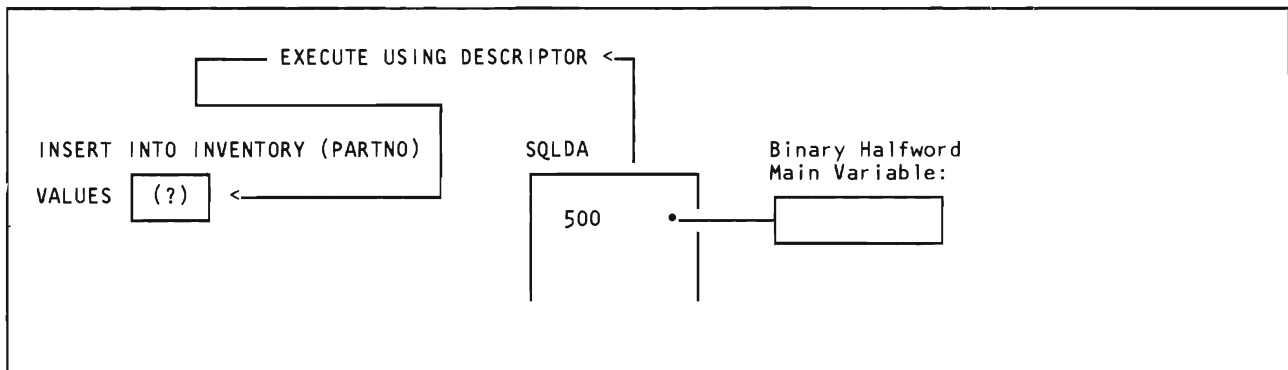


In previous sections, the select-list item, the type code, and the data type of the storage area allocated for holding query results were all equivalent. That is, in the above example, PARTNO is a SMALLINT column (with no nulls permitted), 500 is the type code meaning SMALLINT NOT NULL, and the area allocated is a binary integer halfword. To force a data conversion, you must allocate a storage area having a different data type and then change SQLTYPE in the SQLDA. Suppose that you wanted to select the SMALLINT part numbers into an integer area. Here is the sequence of instructions needed:

```
EXEC SQL PREPARE S1 FROM :STRING
EXEC SQL DESCRIBE S1 INTO SQLDA
Allocate a binary integer fullword of storage.
Set SQLDATA to point to it.
SQLTYPE = 496
```

When the FETCH is executed, SQL/DS performs the SMALLINT to INTEGER conversion automatically. Similarly, you could have converted the retrieved PARTNO values to FLOAT merely by setting SQLTYPE to 480 and by allocating a floating point word of storage.

This conversion can be done when the SQLDA is used for *input* also. Consider the normal case:



As before, PARTNO is SMALLINT. The main variable is also allocated as SMALLINT (binary integer halfword), and the SQLTYPE that describes the main variable represents a SMALLINT. To perform data conversion on input, you need to change only the SQLTYPE and the type of storage allocated to hold the input values. This is done exactly as in the previous example. To insert a floating point variable into the SMALLINT PARTNO column, for example, these steps are needed:

```
EXEC SQL PREPARE S1 FROM :STRING
EXEC SQL DESCRIBE S1 INTO SQLDA
Allocate an 8-byte floating point area.
Set SQLDATA to point to it.
Assign a floating point number to the area.
SQLTYPE = 480
EXEC SQL EXECUTE S1 USING DESCRIPTOR SQLDA
```

All dynamic data conversion is done according to the rules summarized under "Data Conversion" on page 76. Note especially that character to numeric or numeric to character conversions are not allowed.

Should you change the SQLTYPE code and then allocate a storage area of an incorrect type, SQL/DS treats the storage area as though it were of the type indicated by SQLTYPE. For example, suppose SQLTYPE indicates that the storage area pointed to by SQLDATA is an INTEGER, but that the actual area allocated is a binary integer halfword (SMALLINT). SQL/DS treats the field as though it were an INTEGER, not a SMALLINT. This type of error may yield confusing results.

Distressing results are also obtained if SQLTYPE indicates that there is an indicator variable, but you do not allocate one.

The SQL Descriptor Area (SQLDA)

This section summarizes, for your reference, the SQLDA structure and related information.

As you have learned in the previous sections, the SQLDA can be used in any number of ways. In general, the fields within the SQLDA must be initialized either by using a DESCRIBE statement or by user code. Once the fields are initialized, the SQLDA can be used for *input* (in EXECUTE and OPEN) or for *output* (in FETCH).

Figure 21 summarizes the sequence of events needed to initialize the SQLDA for use in processing dynamically defined statements. In any case, you must always initialize SQLN before the DESCRIBE.

Sequence of Events —>				
SQLDA Fields:	First, DESCRIBE initializes:	Then you must initialize:	Next, if you intend to use the SQLDA for input (EXECUTE or OPEN), you must place values in the locations pointed to by SQLDATA and SQLIND. When the SQLDA is used for output (FETCH), SQL/DS will place values in those areas.	EXECUTE, OPEN, and FETCH use:
SQLDAID	X			
SQLDABC	X			
SQLN ¹				X
SQLD	X			X
SQLVAR				
SQLTYPE	X			X
SQLLEN	X			X
SQLDATA		X		X
SQLIND ²		X		X
SQLNAME	X			

Figure 21. SQLDA Initialization

Notes:

1. You must set SQLN before the DESCRIBE.
2. Only provide indicators if they are allowed in the non-dynamic case. (See the previous sections.)

If you do not use a DESCRIBE to set up the SQLDA, you need only fill in those fields that are actually used the OPEN, FETCH, or EXECUTE.

The meanings of the fields within the SQLDA are as follows:

SQLDAID When the SQLDA is used for input or output, SQLDAID does not apply. This field serves only as an SQLDA eye-catcher. It is set to 'SQLDA ' by SQL/DS when a DESCRIBE is first executed. (You never have to fill in SQLDAID.)

SQLDABC When the SQLDA is used for input or output, SQLDABC does not apply. This field is another eye-catcher field. It is set to the length of the SQLDA by SQL/DS when a DESCRIBE is executed. (You never have to fill in SQLDABC.)

SQLN Indicates the number of variables represented by SQLVAR. (SQLN acts as a dimension of the SQLVAR array.) You should always set this value when the structure is allocated. When the USING clause of the DESCRIBE statement is set to NAMES, LABELS, or ANY, you should specify the maximum number of expected select-list items. When you set the USING clause option to BOTH, you should specify twice the number of expected select-list items.

SQLD Indicates the *pertinent* number of elements in the SQLVAR array. When used with a DESCRIBE statement, SQL/DS returns a zero in SQLD if the statement being described is not a SELECT statement. If the statement is a SELECT statement, SQL/DS sets SQLD to indicate the number of SQLVAR elements. The number of SQLVAR elements is either the number of select-list elements (when the USING clause of the DESCRIBE statement is set to NAMES, LABELS, or ANY), or twice the number of select-list elements (if the USING clause is set to BOTH). If (after a DESCRIBE) SQLD is greater than SQLN, the SQLVAR array is not large enough to contain descriptions for all the select-list items. In this case, you must allocate a larger SQLDA based on the value of SQLD.

If you set the value of SQLD yourself, and you set it less than SQLN, the excess elements of the SQLVAR array are ignored.

SQLVAR The individual entries in this array describe the characteristics of dynamically allocated storage areas. These storage areas are intended to hold either the values for "?" parameters (if the SQLDA is used for input) or the values returned from a query (if the SQLDA was used for output). The entries in this array are bound, in order, to the "?" parameters of the prepared statement or to the select-list items (whichever is applicable).

Here is a breakdown of an element of the SQLVAR array; to avoid confusion, keep in mind the distinction between input (OPEN, EXECUTE) and output (DESCRIBE):

SQLTYPE In the case of input, SQLTYPE describes the data type of the allocated storage area and tells whether you are also providing an area for an indicator variable. The data type identified here must be type-compatible with the storage area's use in the prepared statement. (SMALLINT, INTEGER, DECIMAL, and FLOAT are compatible; CHAR, VARCHAR, and LONG VARCHAR are compatible; and GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC are compatible). A method for forcing data conversion was discussed under "Data Conversion" on page 76.

In the case of output, the types are set by SQL/DS to indicate the column types specified in the SELECT-list of the prepared statement.

These are the data codes:

Data Code	Data Type	Indicator Variable?
496	INTEGER	NO
497	INTEGER	YES
500	SMALLINT	NO
501	SMALLINT	YES
484	DECIMAL	NO
485	DECIMAL	YES
480	FLOAT	NO
481	FLOAT	YES
448	VARCHAR	NO
449	VARCHAR	YES
452	CHAR	NO
453	CHAR	YES
456	LONG VARCHAR	NO
457	LONG VARCHAR	YES
468	GRAPHIC	NO
469	GRAPHIC	YES
464	VARGRAPHIC	NO
465	VARGRAPHIC	YES
472	LONG VARGRAPHIC	NO
473	LONG VARGRAPHIC	YES

SQLLEN This field contains the length of the storage area allocated. For DBCS data types, SQLLEN is set to the number of DBCS characters (each DBCS character occupies two bytes in storage). SQLLEN is determined by what is indicated by SQLTYPE.

If SQLTYPE is:	SQLLEN contains:
VARCHAR	the maximum length of the string.
VARGRAPHIC	the maximum number of DBCS characters in the string.
CHAR	the length of the string (fixed).
GRAPHIC	the number of DBCS characters in the string (fixed).
INTEGER	4
SMALLINT	2
FLOAT	8
DECIMAL	precision and scale are in the first (SQLPRCSN) and second (SQLSCALE) bytes, respectively.

SQLDATA This field is never initialized by SQL/DS. You must place in this field a pointer to the storage area that either holds the parameter value (if SQLDA is used for input) or is to hold a select-list result (if the SQLDA is used for output). For varying-length character strings, the actual data should be preceded by a half-word field that specifies the length of the character string. (The value you specify should not include the length of the half-word.) The data must be aligned on a half-word boundary.

SQLIND This field is never initialized by SQL/DS. SQLIND must point to the indicator variable if you have opted to provide one. The indicator variable must be declared as a 15-bit binary integer. If you are using the SQLDA for input, you must provide an appropriate value in the indicator as shown below. (Only null or not null apply to input to SQL/DS.) Note that indicators should not be used when providing input to a WHERE clause. If you are using the SQLDA for output, SQL/DS fills in the indicator using these same values:

0 Denotes that the parameter is not null, and is in the associated storage area.

<0 Denotes that the parameter value is null.

>0 Denotes that a returned value was truncated because the storage area provided was not large enough. If the truncated item was a DBCS or character string, the indicator variable contains the length in characters before truncation. (Applies only for the FETCH statement.)

SQLNAME As indicated in the chart at the beginning of this section, it is never necessary for you to fill in SQLNAME. (SQLNAME is not used in a FETCH, OPEN, or EXECUTE.) When a DESCRIBE is executed, however, SQL/DS fills SQLNAME with information that may be useful in analyzing the select-list items. (Especially when a routine is used to generate a query from a parameterized WHERE clause.)

In general, depending on the option specified in the USING clause of the DESCRIBE statement, either the name or the label associated with the column used in the select-list is returned in positions 1-n of SQLNAME. (The USING clause is described in detail under “DESCRIBE” on page 177.) The exceptions to this are select-list items that are unnamed, such as functions (for example, SUM(SALARIES)), constants ('ABC'), and expressions (A+B-C). In these cases, position 1 of SQLNAME is blank (hexadecimal '40') and positions 3-30 contain a description of the unnamed field. (The value in position 2 varies.) Since a blank (hexadecimal '40') is not allowed in the first byte of SQL identifiers, the application program can tell whether a column name is returned. These rules apply:

Case 1: Basic function. SQLNAME contains the name of the function followed by the column name in parentheses (for example, SUM(SALARIES)). Position 2 of SQLNAME is blank.

Case 2: DISTINCT object of a function. If the keyword DISTINCT is used in the function, it appears before the column name (for example, SUM(DISTINCT SALARIES)). If the column name is large, the whole description may not fit in positions 3-30. In this case, the description is truncated, and hexadecimal 'FF' is set in position 2 of SQLNAME.

Case 3: If the select-list item involves an expression, SQL/DS sets positions 3-n of SQLNAME to this character string:

EXPRESSION *m*

where *m* is a number that identifies the *m*th expression in the select-list. For example, for the sixth expression in the select-list, SQL/DS sets positions 3-n of SQLNAME to EXPRESSION 6. Position 2 of SQLNAME is blank. The above is true for all expressions, even those that contain a built-in function. Expressions include constants, such as 'ABC'.

Case 4: If the object of a function is an expression (for example, SUM(SALARIES+10)), SQL/DS returns in positions 3-n of SQLNAME the name of the function followed by EXPRESSION *m* in parentheses (for example, SUM(EXPRESSION 7)). Position 2 of SQLNAME is blank.

PREPARE

Format:

```
PREPARE statement-name FROM string-spec
```

Examples:

```
PREPARE STAT2 FROM :XSTRING  
PREPARE STAT3 FROM 'DELETE FROM QUOTATIONS WHERE PARTNO = ?'
```

Authorization:

Any user with CONNECT authority can code this statement in an application program and preprocess the program. SQL statements submitted to SQL/DS via the PREPARE and EXECUTE facility have their authorization checked against the privileges of the user who is currently running the program, *not* the user who preprocesses the program.

This statement preprocesses the statement identified by *string-spec* for later execution. String-spec can be either a character constant or a host variable. If string-spec is a host variable, the variable must be declared as fixed- or varying-length character. (If a host variable is used in Assembler or COBOL, it must be varying-length. Fixed-length strings aren't allowed for string-spec in those preprocessors.) String-spec represents a run-time SQL statement.

The "prepared" statement is given the *statement-name* you specify. Statement-name must begin with a letter, \$, #, or @. It can contain up to 18 letters, numbers, \$, #, @, and underscores. Unlike other SQL identifiers, the statement-name must never be enclosed in either single (') or double (") quotes; thus, the statement-name cannot contain embedded blanks. Statement-names can, however, be SQL reserved words. For example:

```
PREPARE SELECT FROM :STRING
```

Note that the statement-name above (SELECT) is *not* enclosed in double quotes. The host variable does not require a colon preceding it; the colon is optional in this statement.

Assembler language programs cannot specify a constant for string-spec. A host variable *must* be used.

The SQL statements you cannot use for string-spec are:

INCLUDE SQLCA	ROLLBACK WORK
INCLUDE SQLDA	COMMIT WORK
WHenever	CONNECT
OPEN	PREPARE
CLOSE	EXECUTE
FETCH	EXECUTE IMMEDIATE
DECLARE CURSOR	DESCRIBE

The SQL statements must not include host language delimiters or contain any references to host variables. If the SQL statement is a SELECT statement, it must not have an INTO clause. (A cursor is used to retrieve results when the statement is executed.)

Although a statement to be “prepared” can not contain any host variables, it can contain parameters to be filled in when the statement is executed. These parameters are denoted by question marks (?). You can specify parameters only in places where a data value could be used. (A parameter can not represent the name of a table or a column.) In the pseudo code example below, an INSERT statement that has two parameters is prepared:

```
QSTRING='INSERT INTO SUPPLIERS(SUPPNO,NAME) VALUES (?,?)'  
PREPARE S1 FROM :QSTRING
```

Each time S1 is executed, values must be supplied for the two parameters that were specified with question marks.

Note that you must supply host language dependent delimiters for the PREPARE statement, but *not* for the statement that is a value in QSTRING. In PL/I the above example is written:

```
QSTRING='INSERT INTO SUPPLIERS (SUPPNO,NAME) VALUES (?,?)';  
EXEC SQL PREPARE S1 FROM :QSTRING;
```

A semicolon is added to the end of the first statement because ordinary PL/I statements are separated by semicolons. The “EXEC SQL” and semicolon on the second statement are the host language delimiters for SQL statements in PL/I.

If your PL/I program constructs dynamic SQL statements by manipulating quoted strings, remember that both SQL and PL/I use two quote marks to represent a single quote mark inside a quoted string. The following PL/I example illustrates this rule:

```
EXEC SQL PREPARE S1 FROM 'INSERT INTO SUPPLIERS(SUPPNO,NAME)  
VALUES (75, 'SMITH')';
```

In this example, the text beginning with INSERT and ending with SMITH'' is considered to be a PL/I constant string. PL/I will collapse each of the quote pairs around SMITH into a single quote before the string is processed by SQL.

In COBOL, a constant string-spec is treated as a COBOL character string and is affected by the QUOTE/APOST option. This option determines the character string delimiters. If you use the same character (' or ') in the constant string-spec as the QUOTE/APOST option establishes for the outer string delimiters, unexpected string termination may result.

It is best to avoid using a constant string-spec whenever it may contain quotes. Instead, you should build the SQL statement as a host variable string-spec, using the known host language rules for character strings. You must be especially careful of SQL statements that contain DBCS constants, because some DBCS characters may contain the encodings for EBCDIC quote. This could cause unintentional termination of host language strings that contain DBCS-type constants.

A question mark can appear in an SQL statement to be "prepared" in any place that a host variable may appear, with the following exceptions:

1. A question mark can not be used in a select-list or FROM-clause (but it may be used in the WHERE clause of a SELECT statement).
2. Two question marks can not appear directly within the same arithmetic or comparison operation: ?+? or ?=? are invalid.
3. If a column name or a literal does not appear to the left of an IN clause, the first item in the list of items to the right of the IN cannot be a ? host variable.
4. There are additional restrictions on the use of ? host variables with hexadecimal literals in comparison predicates. For more information, refer to the description of SQLCODE -422 in the *SQL/Data System Messages and Codes for VM/SP* manual.

The following examples may help clarify when ? variables can and cannot be used. The first set of examples can be successfully processed by SQL/DS PREPARE and EXECUTE statements:

```
UPDATE QUOTATIONS SET QONORDER=?
UPDATE QUOTATIONS SET QONORDER=? WHERE PARTNO+?=?
UPDATE QUOTATIONS SET QONORDER=? WHERE ?+PARTNO=?
UPDATE QUOTATIONS SET QONORDER=? WHERE PARTNO IN (?,?)
UPDATE QUOTATIONS SET QONORDER=? WHERE PARTNO+? IN (?,?)
UPDATE QUOTATIONS SET QONORDER=? WHERE ?+PARTNO IN (?,?)
UPDATE QUOTATIONS SET QONORDER=? WHERE ? IN (0,?)
```

This set of examples cannot be successfully processed by SQL/DS PREPARE and EXECUTE statements:

```
UPDATE QUOTATIONS SET QONORDER=? WHERE PARTNO IN (?)
UPDATE QUOTATIONS SET QONORDER=? WHERE PARTNO=:PART
UPDATE QUOTATIONS SET QONORDER=? WHERE ? IN (?,?)
UPDATE QUOTATIONS SET QONORDER=? WHERE ? IN (?,0)
```

EXECUTE

Format 1:

```
EXECUTE statement-name [USING input-list]
```

Format 2:

```
EXECUTE statement-name [USING DESCRIPTOR input-structure]
```

Examples:

```
EXECUTE S1 USING :X, :Y:YIND  
EXECUTE S1 USING DESCRIPTOR SQLDA  
EXECUTE S1 USING DESCRIPTOR STUFF
```

Authorization:

Any user with CONNECT authority can code this statement in an application program and preprocess the program. SQL statements submitted to SQL/DS via the PREPARE and EXECUTE facility have their authorization checked against the privileges of the user who is currently running the program, *not* the user who preprocessed the program.

Format 1 of the EXECUTE statement causes SQL/DS to execute a statement (identified by *statement-name*) that was “prepared” previously. When the statement is executed, the host variables you list are substituted, in order, into the statement in place of its “?” parameters. Each variable must be of a data type that is compatible with its usage in the “prepared” SQL statement. Each variable can also have an indicator variable if the statement syntax permits them. That is, indicators are permitted for dynamically defined statements if they are permitted in the non-dynamic case. In particular, indicator variables are not allowed in WHERE clauses. All indicator variables must be defined as two-byte integers. A negative value in an indicator variable represents a null value.

You should not execute a dynamically defined statement after ending the logical unit of work in which the statement was prepared. If you do, the results are unpredictable.

You can use PREPARE and EXECUTE to create new objects in the data base. Whenever a new object is created in this manner, the creator or owner of that object is the user who is presently running the program, rather than the user who preprocessed the program. This permits an interactive user at a terminal to create new tables, for example, in the user’s own name rather than in the name of the person who preprocessed the program.

If an error occurs during the execution of an SQL PREPARE statement and the statement name is subsequently executed via an SQL EXECUTE statement, the EXECUTE statement fails.

Format 1 of the EXECUTE statement is used when you know the number and data types of the parameters of the prepared statement. Format 2 permits you to

dynamically specify the “?” parameters of the prepared statement. If you use Format 2, you must use an SQL/DS descriptor (called *SQLDA*) to specify the required parameters. For each variable represented by a “?” in the prepared statement, you must specify information such as length and location in the descriptor. General usage techniques for the *SQLDA* were discussed in earlier sections.

EXECUTE IMMEDIATE

Format:

```
EXECUTE IMMEDIATE string-spec
```

Example:

```
EXECUTE IMMEDIATE :QSTRING
```

Authorization:

Any user with **CONNECT** authority can code this statement in an application program and preprocess the program. SQL statements submitted to SQL/DS via **EXECUTE IMMEDIATE** have their authorization checked against the privileges of the user who is currently running the program, *not* the user who preprocessed the program.

This statement is a short-hand form for preparing and executing SQL statements having no parameters. (See **PREPARE** for string-spec syntax rules.) The statement

```
EXECUTE IMMEDIATE string-spec
```

is exactly equivalent to the following two statements:

```
PREPARE statement-name FROM string-spec
```

```
EXECUTE statement-name
```

EXECUTE IMMEDIATE should be used when the SQL statement is to be executed only once. If a given SQL statement is to be prepared once and executed repeatedly, the non-immediate form of **EXECUTE** should be used.

If string-spec is a host variable, it does not require a colon preceding it; the colon is optional in this statement.

DESCRIBE

Format:

```
DESCRIBE statement-name INTO output-spec  
        [USING {NAMES | LABELS | BOTH | ANY}]
```

Examples:

```
DESCRIBE Q1 INTO SQLDA  
DESCRIBE S1 INTO STR1
```

Authorization:

You can use DESCRIBE for any statement you have successfully prepared.

The DESCRIBE statement obtains information about a statement that has been prepared. Structure-spec should name an SQLDA structure. If the prepared statement is a SELECT statement, DESCRIBE returns the number of fields in the answer set, and the data types, lengths, and names of these fields. If the prepared statement is not a SELECT statement, DESCRIBE sets the SQLDA field called SQLD to zero.

All fields in the SQLDA were described under “The SQL Descriptor Area (SQLDA)” on page 167. General usage techniques are described under “Dynamically Defined Queries” on page 151.

You should not attempt to DESCRIBE a statement that was prepared in a different logical unit of work. If you do, the results are unpredictable.

The USING clause can be used to return column labels. You can specify one of four parameters with the USING clause to tell SQL/DS which values to return in the SQLNAME field of the SQLDA. The NAMES parameter is the default. It tells SQL/DS to return column names but no column labels.

The LABELS parameter tells SQL/DS to return column labels but no column names. The BOTH parameter specifies that both column labels and column names are to be returned. In this case, the value returned in SQLDA is twice the number of columns (N) in the select-list. The values returned in SQLVAR elements are as follows:

1. Elements 1 through N

The same as when only column names are returned.

2. Elements N+1 through 2N

- a. SQLTYPE: 0
- b. SQLLEN: 0

- c. SQLDATA: 0
- d. SQLIND: 0
- e. SQLNAME: column label

Column labels are given in a sequence which corresponds with the sequence in which column names are given in the first N SQLVAR elements.

If a label exists for a column, the ANY parameter of the USING clause tells SQL/DS to return it in the SQLNAME field. If not, the column name is returned. A label is considered not to exist if the length of the label is zero or if the value of the label is null.

If either LABELS or BOTH is specified in the USING clause and a label does not exist, SQLNAME is set to a length of zero, and the field is cleared to 30 blanks. Therefore, when either LABELS or BOTH is specified, your program must be prepared to receive a zero-length label in the SQLNAME field. That is, if you wish to move a label from an SQLNAME field into a user work area using the length returned in the SQLDA, you must first make sure that the length is not zero.

While column names cannot start with a blank, column labels can start with anything. Therefore, the program cannot tell whether the select-list element is a built-in function, an expression, or a label. The ANY option should be avoided on a data base where this situation might arise.

If the described SELECT statement contains a union, column labels of the first SELECT are returned.

If the select-list contains a built-in function, the label is returned in SQLNAME with the built-in function, as it is for a column name. However, unlike for a column name, the first two bytes are not used as the flag bytes (hexadecimal '4040' or '40FF') for a label. That is, all thirty bytes are used for the built-in function and the label.

If a literal is used in the select-list when LABELS is specified, the literal is treated as a nonexistent column label. If either NAMES or ANY is specified, a two byte flag plus "EXPRESSION m" is returned. If BOTH is specified, the two byte flag plus "EXPRESSION m" is returned as the name, and the label is treated as nonexistent.

If the DBCS option is set to YES, SO/SI pairing is guaranteed in SQLNAME, not only when truncation occurs but also when the original value has an un-matching SO for both column names and column labels.

DECLARE CURSOR Statement for Dynamically Defined Queries

Format:

```
DECLARE cursor-name CURSOR FOR statement-name
```

Example:

```
DECLARE CUR10 CURSOR FOR STAT1
```

Before you can execute a “prepared” SELECT or INSERT statement and fetch or insert its results, you must declare a cursor. For dynamically defined SELECT or INSERT statements, you must use the variation of the SQL DECLARE CURSOR statement that is shown above.

Cursor-name must begin with a letter, \$, #, or @. It can contain up to 18 letters, numbers, \$, #, @, and underscores. Unlike other SQL identifiers, the cursor-name must never be enclosed in either single (') or double (") quotes; thus, the cursor-name cannot contain embedded blanks. Cursor-names can, however, be SQL reserved words. For example:

```
DECLARE UPDATE CURSOR FOR STATM1
```

Note that the cursor-name above (UPDATE) is *not* enclosed in double quotes.

Cursor names must be unique in the same logical unit of work. If they are not, an error will result. If a cursor name is the same as a statement-name in the same access module, unpredictable results may occur.

Here is another example of a query-cursor declaration that associates cursor C1 with a SELECT statement called QUERY1:

```
DECLARE C1 CURSOR FOR QUERY1
```

QUERY1 must be “prepared” before the cursor is declared. The actual retrieval of result rows is shown in Figure 22.

```
EXEC SQL OPEN C1
EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
DO WHILE (SQLCODE = 0)
    DISPLAY (results pointed to by SQLDATA and SQLIND
            for all pertinent SQLVAR elements)
    EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE C1
```

Figure 22. Using a Cursor with Dynamically Defined Statements

You should not attempt to declare a cursor for a statement that was prepared in a different logical unit of work. If you do, the results are unpredictable.

Refer to “Dynamically Defined Queries” on page 151 if you need more information about processing a run time query.

OPEN Statement with USING Option

Format 1:

```
OPEN cursor-name [USING host-variable-list]
```

Format 2:

```
OPEN cursor-name [USING DESCRIPTOR structure-spec]
```

Examples:

```
OPEN C1 USING :X, :Y  
OPEN C2 USING DESCRIPTOR SQLDA  
OPEN C3
```

An option on the OPEN statement allows you to open a cursor on a “prepared” SELECT statement, and to bind the values of the “?” parameters. For example, suppose statement S1 is prepared using the following query:

```
SELECT PRICE FROM QUOTATIONS WHERE PARTNO=? AND SUPPNO=?
```

When you open a cursor to fetch the results of the query, you must provide two variables that supply the missing part number and supplier number. You can do this by listing host variables (Format 1) or by allocating a suitable SQLDA structure (Format 2). You can not use indicator variables in the OPEN statement. The use of the SQLDA structure is described in earlier sections.

To change the values of the host variables and hence the active set, you must close and re-open the query. (This does not cause the query to be “prepared” again, however.)

Note: When you are opening an insert-cursor, you should not specify the USING option.

FETCH Statement for Dynamically Defined Queries

Format 1:

```
FETCH cursor-name INTO host-variable-list
```

Format 2:

```
FETCH cursor-name USING DESCRIPTOR structure-spec
```

Examples:

```
FETCH C1 INTO :X, :Y:YIND  
FETCH C1 USING DESCRIPTOR SQLDA
```

The FETCH statement retrieves one row of a query result defined by a PREPARE statement. The indicated cursor must be declared and opened. The places into which the individual fields are to be fetched are indicated by a list of host variables (optionally with indicator variables for null values), or by the SQLDA. General usage techniques for the SQLDA were described in earlier sections. If no rows remain in the active set of the cursor used in a FETCH statement, the “not found” condition (SQLCODE=100) is returned.

PUT Statement for Dynamically Defined Inserts

Format 1:

```
PUT cursor-name FROM host-variable-list
```

Format 2:

```
PUT cursor-name USING DESCRIPTOR structure-spec
```

Examples:

```
PUT C1 FROM :X, :Y  
PUT C1 USING DESCRIPTOR SQLDA
```

The PUT statement inserts one row of data defined by a PREPARE statement. The indicated cursor must be declared and opened. The sources of the data to be inserted are indicated by a list of host variables or by the SQLDA. The host variables or the SQLDA supply values for ? parameters in the INSERT statement that was either prepared or defined with a DECLARE CURSOR. General usage techniques for the SQLDA were described in earlier sections.



Preprocessing and Running the Program

Contents

Introduction	184
VM/SP Connect Considerations	186
Initializing Your User Machine	186
Preprocessing the Program	187
Compiling the Program	196
Link-Editing and Loading the Program	197
Running your Program	198
Multiple User Mode	198
Single User Mode	199
Specifying User Parameters	199

Introduction

This section discusses considerations for running application programs that access SQL/DS. Topics described in this section are:

- Initializing a user machine
- Preprocessing
- Compiling
- Loading
- Executing

You can run applications in either single user mode or multiple user mode. In either case, you must have access to the SQL/DS production (Q) minidisk. Refer to the *SQL/Data System Installation for VM/SP* and *SQL/Data System Planning and Administration for VM/SP* manuals. In addition, you may have a choice of SQL/DS data bases in which to preprocess and run your program.

1. Single User Mode

In single user mode, SQL/DS, its preprocessors, and your application programs run in a single virtual machine. A parameter in the SQL/DS startup EXEC, SQLSTART, defines this mode (SYSMODE=S). Because both SQL/DS and the user application run in the same virtual machine, single user mode is sometimes referred to as single virtual machine mode.

2. Multiple User Mode

In multiple user mode, one or more users or applications concurrently access the same data base. For this mode of operation, SQL/DS runs in a virtual machine while one or more SQL/DS application programs or preprocessors operate in other virtual machines. Multiple user mode is sometimes referred to as multiple virtual machine mode. This mode is defined by the initialization parameter, SYSMODE=M.

3. Multiple Data Base Operation

Multiple data base operation refers to operating more than one SQL/DS data base machine in multiple user mode on the same VM/SP system. This implies multiple data bases being accessed concurrently by many users. When starting SQL/DS in single user mode, you may also have a choice of data bases; however, in single user mode, the data base is not shared and you may not change it without restarting SQL/DS.

4. SQL/DS Data Base Machine

The SQL/DS data base machine is a VM/SP virtual machine that owns minidisks that make up one or more SQL/DS data bases (each data base has an assigned name). A data base machine is active for only one data base at a

time; the SQL/DS data base machine is initiated by an SQLSTART EXEC and terminated by an SQLEND operator command. The SQLSTART EXEC not only has a parameter for the DBNAME (SQL/DS data base), but also has a parameter for the mode (multiple or single user). Once a data base machine has been activated in multiple user mode, it is possible for multiple users to access the SQL/DS data base. In order to do this users normally must have:

- Proper SQL/DS authorization
- A VM/SP IUCV path to the data base machine
- Read access to the SQL/DS production disk
- Executed the SQLINIT EXEC at some time, as described below. This establishes the current data base association.

These modes of operation are illustrated in Figure 23.

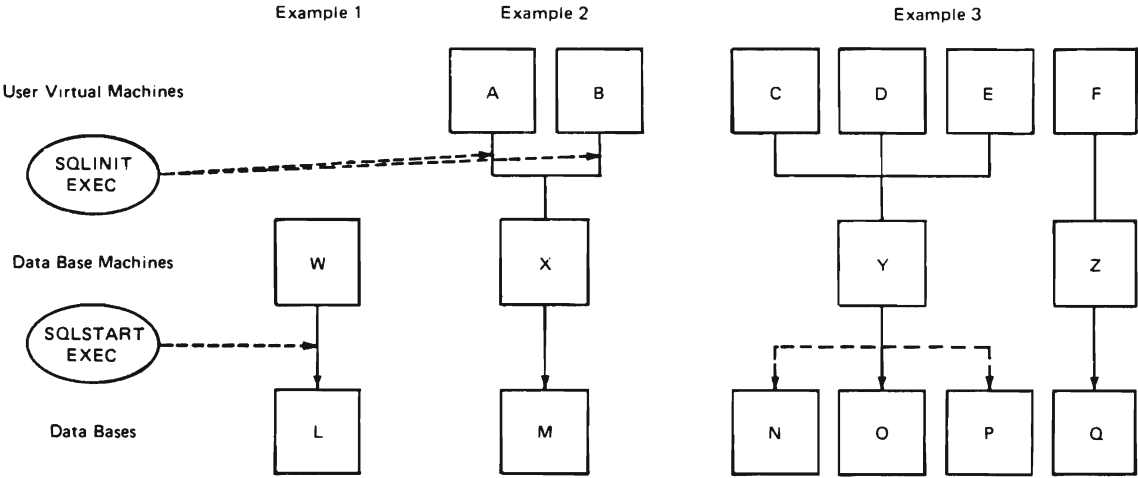


Figure 23. SQL/DS Modes of Operation

In the first example, an SQLSTART EXEC has activated SQL/DS data base L in single user mode (W is the data base machine for data base L).

In the second example, an SQLSTART EXEC has activated the SQL/DS data base M in multiple user mode (X is the data base machine for data base M). User virtual machines A and B have used the SQLINIT EXEC to select M as their SQL/DS data base.

In the third example, SQLSTART EXECs have started data base O from data base machine Y and data base Q from data base machine Z. Note that Y also owns data bases N and P, but only one data base may be activated concurrently. Users in virtual machines C, D, E, and F may choose between the two active data bases (O and Q) via an SQLINIT EXEC. Currently, only one user (F) has chosen data base Q; the remainder have chosen data base O.

VM/SP Connect Considerations

Although explicit CONNECT is supported in the VM/SP environment, it is not required. When a CONNECT statement is omitted, SQL/DS accepts the password verification of the VM/SP virtual machine and uses the VM/SP userid as the SQL/DS userid. This SQL/DS support is called "implicit connect." Implicit connect requires one of the following authorizations:

1. The special userid "ALLUSERS" must have been granted CONNECT authority, or
2. The individual users must have been granted CONNECT authority.

Passwords are not necessary in the GRANT commands that establish the above connect authorizations (passwords are necessary only when explicit connects are required).

For example, assume the following GRANT command:

```
GRANT CONNECT TO A, B, C, ALLUSERS
```

After this command, any VM/SP user may be implicitly connected to SQL/DS. However, if the following command is used:

```
REVOKE CONNECT FROM ALLUSERS
```

only users A, B, C can be implicitly connected to SQL/DS.

Thus the special userid "ALLUSERS" can be used to selectively turn the implicit connect capability on or off for the total user set, while individual users can retain the implicit connect privilege.

SQL/DS in a VM/SP environment supports multiple data base machines and access to multiple data bases. From an application program standpoint, the connect function is the final step in the process of linking to a particular SQL/DS data base. Provision for specifying a data base machine is not included in the CONNECT statement since it is determined by user or operator actions prior to running the program. This is discussed in the *SQL/Data System Planning and Administration for VM/SP* manual.

Initializing Your User Machine

Format:

```
SQLINIT [Dbname (dbname) [dcssID (dcss-id) ]]
```

Example:

```
SQLINIT DBNAME(SQLDBA)
```

The parameters of SQLINIT are as follows:

Dbname(dbname)

DBNAME identifies the SQL/DS data base to be accessed. DBNAME is an optional parameter; if no entry is provided, SQLDBA is the default data base. The abbreviation for DBNAME is D.

dcssID(dcss-id)

DCSSID is an optional parameter. It should be specified only if there are discontinuous saved segments for the SQL/DS code. The abbreviation for DCSSID is ID.

When you preprocess a program or run an SQL/DS application program in multiple user mode, you must first establish the data base that you want the program to access. This is done by using a VM/SP EXEC called SQLINIT. It is provided by SQL/DS. You invoke this EXEC by logging on to your user virtual machine, IPLing CMS, and invoking the EXEC. You need only to do this once, as long as you continue to operate the same data base. Even if you log off and back on to your virtual machine, you retain your association with the data base that was established by the SQLINIT EXEC (the association is recorded on your A-disk). The only exception to this is when the data base machine that is associated with the data base is changed. If you decide to change to a different data base, you must use the SQLINIT EXEC again, specifying the new data base.

For additional information on the SQLINIT EXEC, refer to the *SQL/Data System Planning and Administration for VM/SP* manual.

Preprocessing the Program

Once a data base is established, you may preprocess programs that use the data base by invoking the VM/SP EXEC, SQLPREP, that is provided by SQL/DS. In multiple user mode, the data base machine that owns the selected data base must have been started. This is normally a function of a data base administrator or their representative.

In single user mode, you also use the SQLPREP EXEC, but in this case, it does more work for you. It establishes an SQL/DS data base machine in your virtual machine, where you are the sole user of the data base. You must choose the data base you want to access, through the DBNAME= parameter. In fact, when you specify the data base name as a parameter to the SQLPREP EXEC, you are also indicating that you want to run in single user mode. The SQLPREP EXEC does the SQLSTART for you.

For additional information, refer to the Operations Chapter in the *SQL/Data System Planning and Administration for VM/SP* manual.

There are four preprocessors supplied with SQL/DS. They have the following program names:

PLI -- PL/I Preprocessor

ASM -- Assembler Preprocessor

COBOL -- COBOL Preprocessor

FORTRAN -- FORTRAN Preprocessor

The preprocessor takes SYSIN source program input and produces a modified source program. The modified source program output is put to SYSPUNCH and printed output is put to SYSPRINT. The SQLPREP EXEC allows you to direct SYSIN, SYSPUNCH, and SYSPRINT to various virtual devices and CMS files.

If the preprocessor encounters a severe error in an SQL statement, only syntactic checking is performed on subsequent SQL statements. (That is, all errors may not be found on the first pass.) If successful, the preprocessor places an entry in the catalog SYSTEM.SYSACCESS to record the newly-created access module.

The format for the SQLPREP EXEC follows:

Format:

```
SQLPREP { PLI | COBo1 | ASM | FORTran }

  PrepParm( PREPname=program-name
            [ ,USERid=userid/password ]
            [ { ,KEEP | ,REVOKE } ]
            [ { ,NOCHECK | ,CHECK } ]
            [ { ,NOGRaphic | ,GRaphic } ]      (for PL/I and COBOL only)
            [ { ,PRINT | ,NOPrint } ]
            [ { ,PUnch | ,NOPUnch } ]
            [ { ,Quote | ,APOST } ]          (for COBOL only)
            [ ,LineCount(integer) ]
            [ ,ISOLation( RR | CS | USER ) ]
            [ { ,BLoCK | ,NOBLoCK } ] )

[ sysIN( { filename [ filetype [ filemode ] ] | Reader } ) ]

[ sysPRINT( { filename [ filetype [ filemode ] ] | Printer |
            Terminal } ) ]

[ sysPUnch( { filename [ filetype [ filemode ] ] | Punch } ) ]

[ Dbname(dbname)                (Note: Specify for
  [ dcSSID(dcSS-id) ]             Single User Mode only)
  [ LOGmode ( Y | A | N ) ]
  [ PARMID(filename) ] ]
```

Note: Abbreviations for keywords are in upper-case letters.

Examples:

Single User Mode:

```
SQLPREP COB PP(PREP=MYJOB,USERID=JERRY/SECRET,KEEP,CHECK,
PRINT,PUNCH,LC(66)) DBNAME(MYDB) IN(MYINPUT) LOG(A)
```

Multiple User Mode:

```
SQLPREP COB PP(PREP=MYJOB,USERID=JERRY/SECRET,KEEP,CHECK,
PRINT,PUNCH,LC(66)) IN(MYINPUT)
```

The parameters of the SQLPREP EXEC are:

PLI | COBOL | ASM | FORTRAN

This parameter identifies to the EXEC which preprocessor is to be executed. This parameter is required and *must be specified first*. The abbreviation for COBOL is COB and FORTRAN is FORT. The order in which you specify the other keywords is not important.

PREPPARM

PrepParm(PREPname=program-name
[,USERid=userid/password]
[{ ,KEEP | ,REVOKE }]
[{ ,NOCHECK | ,CHECK }]
[{ ,NOGRaphic | ,GRaphic }] (for PL/I and COBOL only)
[{ ,PPrint | ,NOPPrint }]
[{ ,PUnch | ,NOPUnch }]
[{ ,Quote | ,APOST }] (for COBOL only)
[,LineCount(integer)]
[,ISOLation(RR | CS | USER)]
[{ ,BLoCK | ,NOBLoCK }])

These parameters are used to specify the preprocessor options. The abbreviation for PREPPARM is PP.

PREPNAME=program-name

program-name is the name by which SQL/DS will know the access module. The length of program-name is limited to eight characters. In addition, program-name cannot contain the character #. Otherwise, program-name follows the rules for formulating table and column names (as described under “General Rules for Naming Data Objects” on page 74. PREPNAME is a required parameter and can be abbreviated PREP.

USERID=userid/password

userid identifies the creator of the access module to the SQL/DS data base. The password, if specified, should agree with the one established for this userid by an SQL/DS GRANT statement. This information may be used in executing a CONNECT statement to gain access to the SQL/DS data base which will determine if proper authorization exists for the SQL/DS statements contained in the program. The abbreviation for USERID is USER.

If the USERID option is not specified, then the VM logon userid will be used as the *creator* of the access module to the SQL/DS data base. The VM logon userid will be implicitly connected to the SQL/DS data base. All the SQL/DS statements in the program will have their authorization checked against the implicitly connected userid.

KEEP | REVOKE

These parameters are applicable if this program has previously been preprocessed, and the creator has granted the RUN privilege on the resulting access module to some other users. The KEEP parameter causes these grants of the RUN privilege to remain in effect when the preprocessor produces the new access module. If you specify the REVOKE parameter, or if the creator of the program is not entitled to grant all the privileges embodied in the program, the preprocessor removes all existing grants of the RUN privilege. KEEP and REVOKE are optional; the default is KEEP. KEEP and REVOKE have no abbreviations.

CHECK | NOCHECK

When the NOCHECK parameter is specified, this causes the preprocessor to execute normally, that is, to generate modified source code and perform access module functions. If CHECK is specified, this parameter causes the preprocessor to check all SQL statements for validity and generate error messages if necessary. However, the preprocessor does not generate an access module or modified source code. CHECK and NOCHECK are optional parameters; the default is NOCHECK. CHECK and NOCHECK have no abbreviations.

GRAPHIC | NOGRAPHIC (PL/I and COBOL preprocessors only)

GRAPHIC indicates that the preprocessor should scan for DBCS constants in the source statements and comments, including SQL/DS statements. GRAPHIC *must* be specified if DBCS data is used in the host language or SQL statements. GRAPHIC and NOGRAPHIC are optional parameters; NOGRAPHIC is the default. The abbreviation for GRAPHIC is GR and NOGRAPHIC is NOGR.

PRINT | NOPRINT

When the NOPRINT parameter is specified, this causes the preprocessor listing output to be suppressed except for the summary messages which are normally printed at the end of the preprocessor listing output. PRINT specifies that the preprocessor modified source listing output, including the summary messages, should be produced. PRINT and NOPRINT are optional parameters; PRINT is the default. The abbreviation for PRINT is PR and NOPRINT is NOPR.

PUNCH | NOPUNCH

When the NOPUNCH parameter is specified, this causes the preprocessor modified source output to be suppressed. PUNCH specifies that the preprocessor modified source output should be produced. PUNCH and NOPUNCH are optional parameters; PUNCH is the default. The abbreviation for PUNCH is PU and NOPUNCH is NOPU.

QUOTE | APOST (COBOL preprocessor only)

The QUOTE preprocessor parameter should be used whenever the QUOTE parameter is used in the COBOL compiler.

QUOTE causes the preprocessor to use double quotes (") as constant delimiters in the VALUE clauses of the declarations it generates. The use of a single-quote (') or a double-quote (") in SQL statements is not affected by this parameter. If you do not specify this parameter, the COBOL preprocessor defaults to APOST and generates apostrophes or single-quote (') delimiters for its internal source declarations. The abbreviation for QUOTE is Q.

LINECOUNT(integer)

The LINECOUNT option allows you to specify how many lines per page are to be printed by the preprocessor in the output listing. The value, *integer* specifies the number of lines per page value. The valid range for this value is 10 to 32767. If no value is specified, or if there is an error in the specification of the LINECOUNT parameter, then

the default LINECOUNT value of 60 is used. The abbreviation for LINECOUNT is LC.

ISOLATION (**RR** | **CS** | **USER**)

The ISOLATION option allows you to specify the isolation level that your program will run at. If RR (repeatable read) is specified, SQL/DS will hold a lock on all data read by the program in the current logical unit of work. If CS (cursor stability) is specified, SQL/DS will only hold a lock on the row or page of data pointed to by a cursor. If USER is specified, the application program will control its isolation level. The default is RR. The abbreviation for ISOLATION is ISOL. See "Selecting the Isolation Level" on page 251 for guidelines on choosing the isolation level for your program.

BLOCK | **NOBLOCK**

When the BLOCK parameter is specified, SQL/DS inserts and retrieves rows in groups. This improves performance for programs running in multiple user mode where many rows will be inserted or retrieved. NOBLOCK tells SQL/DS not to group rows. The abbreviation for BLOCK is BLK; for NOBLOCK it is NOBLK. The default is NOBLOCK. BLOCK and NOBLOCK are optional parameters. See "To Block or Not to Block?" on page 254 for guidelines on deciding which programs to specify blocking for.

Assume, for example, you wanted to specify these preprocessor parameters: prepname = SAMPLE1, userid = USER1, password = PW, NOPRINT, KEEP, and default values for the remaining options. The following parameters would be used:

```
PREPPARM (PREP=SAMPLE1 , USER=USER1/PW , NOPRINT , KEEP)
```

Now, if you wanted to have a linecount of 70 and REVOKE instead of KEEP the command would look like this:

```
PREPPARM (PREP=SAMPLE1 , USER=USER1/PW , NOPRINT , REVOKE , LC (70) )
```

SYSIN

Two choices exist:

1. SYSIN(filename [filetype [filemode]])

This optional parameter identifies the *filename* (fn) and optionally the *filetype* (ft) and *filemode* (fm) of the CMS file containing the preprocessor source input. The filetype specification will default to the following:

PLI -- "PLISQL "

COBOL -- "COBSQL "

ASM -- "ASMSQL "

FORTRAN -- "FORTSQL "

The filemode specification will default to A.

If this form of the SYSIN parameter is supplied, the following CMS FILEDEF command is issued for the preprocessor source input file:

```
FILEDEF SYSIN DISK fn ft fm (RECFM FB LRECL 80 BLOCK 800
```

2. SYSIN(READER)

This specification of the SYSIN optional parameter identifies that the preprocessor source input file is a virtual reader file. If SYSIN(READER) is specified, the following CMS FILEDEF command is issued for the preprocessor source input file:

```
FILEDEF SYSIN READER (RECFM F LRECL 80
```

The abbreviation for READER is: R, RE, REA, READ, or READE.

If the SYSIN information is not specified, then the user must issue a CMS FILEDEF command for the preprocessor source input (ddname=SYSIN) before the SQLPREP EXEC is issued. The abbreviation for SYSIN is IN.

SYSPRINT

Three choices exist:

1. SYSPRINT(filename [filetype [filemode]])

This optional parameter identifies the *filename* (fn) and optionally the *filetype* (ft) and *filemode* (fm) of the CMS file containing the preprocessor source output listing. The filetype specification will default to LISTPREP and the filemode specification will default to A.

If this form of the SYSPRINT parameter is supplied, the following CMS FILEDEF command is issued for the preprocessor source output listing file:

```
FILEDEF SYSPRINT DISK fn ft fm . . .  
                (RECFM FBA LRECL 121 BLOCK 1210
```

2. SYSPRINT(PRINTER)

This specification of the SYSPRINT optional parameter identifies that the preprocessor source output listing file is directed to a virtual printer file. If SYSPRINT(PRINTER) is specified, the following CMS FILEDEF command is issued for the preprocessor source output listing file:

```
FILEDEF SYSPRINT PRINTER (RECFM FA LRECL 121
```

The abbreviation for PRINTER is: P, PR, PRI, PRIN, PRINT, or PRINTE.

3. SYSPRINT(TERMINAL)

This specification of the SYSPRINT optional parameter identifies that the preprocessor source output listing file is directed to the console terminal. If SYSPRINT(TERMINAL) is specified, the following CMS FILEDEF command is issued for the preprocessor source output listing file:

```
FILEDEF SYSPRINT TERM (RECFM FA LRECL 121
```

The abbreviation for TERMINAL is: T, TE, TER, TERM, TERMI, TERMIN, or TERMINA.

If the SYSPRINT information is not specified and the preprocessor source input file was assigned to the virtual reader, then the preprocessor source output listing file is assigned to the virtual printer via the CMS FILEDEF command described above.

If the SYSPRINT parameter is not specified and the preprocessor source input file was assigned to a CMS file, then the following default CMS FILEDEF command is issued for the preprocessor source output listing file:

```
FILEDEF SYSPRINT DISK fn LISTPREP A . . .  
      (RECFM FBA LRECL 121 BLOCK 1210
```

where:

fn is the filename specification used for the preprocessor source input file and filemode is defaulted to A.

If neither SYSIN nor SYSPRINT information is specified, then the user must issue a CMS FILEDEF command for the preprocessor source output listing file (ddname=SYSPRINT) before the SQLPREP EXEC is issued. The abbreviation for SYSPRINT is PR.

SYSPUNCH

Two choices exist:

1. SYSPUNCH(filename [filetype [filemode]])

This optional parameter identifies the *filename* (fn) and optionally the *filetype* (ft) and *filemode* (fm) of the CMS file containing the preprocessor modified source output. The filetype specification will default to a value based on the preprocessor invoked, as follows:

PLI -- "PLIOPT "

COBOL -- "COBOL "

ASM -- "ASSEMBLE"

FORTRAN -- "FORTRAN "

The filemode specification will default to A.

If this form of the SYSPUNCH parameter is supplied, the following CMS FILEDEF command is issued for the preprocessor modified source output file:

```
FILEDEF SYSPUNCH DISK fn ft fm . . .  
          (RECFM FB LRECL 80 BLOCK 800
```

2. SYSPUNCH(PUNCH)

This specification of the SYSPUNCH optional parameter identifies that the preprocessor modified source output file is to a virtual punch file. If SYSPUNCH(PUNCH) is specified, the following CMS FILEDEF command is issued for the preprocessor modified source output file:

```
FILEDEF SYSPUNCH PUNCH (RECFM F LRECL 80
```

The abbreviation for PUNCH is: P, PU, PUN, or PUNC.

If the SYSPUNCH information is not specified and the preprocessor source input file was assigned to the virtual reader, then the preprocessor modified source output file is assigned to the virtual punch via the CMS FILEDEF command described above.

If the SYSPUNCH parameter is not specified and the preprocessor source input file was assigned to a CMS file, then the following default CMS FILEDEF command is issued for the preprocessor modified source output file:

```
FILEDEF SYSPUNCH DISK fn ft A . . .  
          (RECFM FB LRECL 80 BLOCK 800
```

where:

fn is the filename specification used for the preprocessor source input file and filemode is defaulted to A. *ft* is the default filetype as determined by the previously mentioned method.

If neither SYSIN nor SYSPUNCH information is specified, then the user must issue a CMS FILEDEF command for the preprocessor modified source output file (ddname=SYSPUNCH) before the SQLPREP EXEC is issued. The abbreviation for SYSPUNCH is PU.

DBNAME(dbname)

This parameter indicates that the preprocessor being invoked is to execute in SQL/DS single user mode. It also identifies the name of the SQL/DS data base to be accessed by the SQL statements contained in the preprocessor source input file.

If this parameter is specified, it will be used as the DBNAME parameter for the SQLSTART EXEC that is executed for you to start up SQL/DS in single user mode. The SQL/DS system initialization parameters SYSMODE=S, and PROGRAM=progrname (where progrname varies according to which preprocessor is being invoked) will also be supplied in the PARM parameter of the SQLSTART EXEC. The abbreviation for DBNAME is: D, DB, DBN, DBNA, or DBNAM.

DCSSID(dcssid)

This applies only when running a preprocessor in SQL/DS single user mode. The parameter can only be specified if the DBNAME parameter is also specified. This parameter identifies the method in which all SQL/DS System modules will be loaded for execution.

If this parameter is specified, it will be used as the DCSSID parameter for the SQLSTART EXEC. If this parameter is omitted, then no DCSSID parameter will be passed to the SQLSTART EXEC. The abbreviation for DCSSID is ID.

Refer to the *SQL/Data System Planning and Administration for VM/SP* manual for a further description of the SQL/DS System DCSSID parameter.

LOGMODE(Y | A | N)

This applies only when running a preprocessor in SQL/DS single user mode. The parameter can only be specified if the DBNAME parameter is also specified. It identifies the value to be used for the SQL/DS system initialization LOGMODE parameter when SQL/DS is started in single user mode.

If this parameter is omitted and the DBNAME parameter is specified, the LOGMODE parameter will not be supplied as an SQL/DS system initialization parameter in the SQLSTART EXEC. The abbreviation for LOGMODE is LOG.

Refer to the *SQL/Data System Planning and Administration for VM/SP* manual for a further description of the SQL/DS System LOGMODE parameter and the log mode considerations.

PARMID(filename)

This applies only when running a preprocessor in SQL/DS single user mode. The parameter can only be specified if the DBNAME parameter is also specified. This parameter identifies filename of a CMS file that contains SQL/DS initialization parameters.

If this parameter is omitted and the DBNAME parameter is used, the PARMID parameter will not be passed as an SQL/DS initialization parameter to the SQLSTART EXEC. PARMID has no abbreviation.

Refer to the *SQL/Data System Planning and Administration for VM/SP* manual for a further description of the SQL/DS System PARMID parameter.

Compiling the Program

Once you have successfully preprocessed your program, you can use the standard compilers to create an object code program. Use the modified source output from the SQL/DS preprocessor as input to the compiler. Except as noted below, there are no extra compiler options or procedures necessary for compiling preprocessed SQL/DS application programs.

If your PL/I application program contains DBCS data, you must specify the GRAPHIC option for the compiler. If your COBOL application program contains DBCS data, the output of the SQL/DS preprocessor must be processed by the COBOL Kanji Processing function of OS/VS Utility Program -- Kanji, Program Product number 5799-BBA, RPQ reference number 7F0095.

If the QUOTE option is used for the SQL/DS COBOL preprocessor, it should also be used for the COBOL compiler.

Link-Editing and Loading the Program

After compilation, programs must be loaded before they can be executed. When loading *any* SQL/DS application, you must link-edit extra SQL/DS TEXT file(s). The resource manager stub is one TEXT file that must be link-edited with all SQL/DS application programs. The application program communicates with SQL/DS through the resource manager stub.

The resource manager stub routine has a filename of ARIRVSTC. This stub routine is invoked, however, by its CSECT name ARIPRDI. To link-edit this stub routine successfully with the user program, you must INCLUDE ARIRVSTC or place the TEXT files in a CMS TXTLIB, which will make the entry point ARIPRDI known to the link-edit process.

To INCLUDE a TEXT file, place the included TEXT filename(s) after the user's program name in the CMS LOAD command. Note that the CMS LOAD command will automatically load the needed TEXT files if the user machine has READ access to the production minidisk. That is, the CMS LOAD command automatically searches all accessed CMS minidisks in ascending order (A through Z) for TEXT files that it needs. For additional information about CMS LOAD, see the *VM/SP CMS Command and Macro Reference* manual.

A way to avoid specifying ARIRVSTC in the CMS LOAD command is to put ARIRVSTC and all your application TEXT files into a TXTLIB. To create a TXTLIB, issue the following command:

```
TXTLIB GEN my-lib ARIRVSTC program-name . . .
```

To add new programs to a TXTLIB, issue the following command:

```
TXTLIB ADD my-lib program-name2 program-name3 . . .
```

Once a program is in a TXTLIB, issue the following commands to perform the link-edit:

```
GLOBAL TXTLIB my-lib  
LOAD program-name
```

For more information about TXTLIB, see the *VM/SP CMS Command and Macro Reference* manual.

- For all programs written in FORTRAN, you must also link-edit the TEXT file ARIPEIFA.

- For all programs written in COBOL, you must also link-edit the TEXT file ARIPADR.
- For all programs that include the DBS utility, you must also link-edit the TEXT files ARISYSDC, ARIDSQLA, and ARIDDFP.

All of these TEXT files are on the SQL/DS production minidisk (Q-disk). After loading the SQL/DS application, you should create a module by issuing the CMS GENMOD command. This module can be used in multiple user mode, but is not required; it is required, however, to run in single user mode. For example, if you wanted to create a module for a SQL/DS Assembler application program called SAMPLE1 that has been compiled and added to a TXTLIB called LIBRARY1, you would issue the following commands:

```
GLOBAL TXTLIB LIBRARY1
LOAD SAMPLE1
GENMOD SAMPLE1
```

After these commands have been issued, a CMS file with a filename of SAMPLE1 and a filetype of MODULE is created.

Running your Program

How you execute SQL/DS applications varies according to the mode in which SQL/DS is running.

Multiple User Mode

When SQL/DS has been started in multiple user mode, the user machine should have IPLed CMS and been initialized (via the SQLINIT EXEC).

File definitions may be required if the program has any input or output files. The CMS FILEDEF command is described in the *VM/SP CMS Command and Macro Reference* manual.

If a module was created, you can execute the program by specifying the name of the module followed by any user program parameters. For example, the following illustrates starting an Assembler program named SAMPLE1 in multiple user mode. The user parameters are passed directly to the program:

```
SAMPLE1 parm1 parm2
```

If a module was not created, you can execute the program by specifying the CMS LOAD command, as described in the previous section, then specify the CMS START command. For example, we can execute the previous program named SAMPLE1 with the following commands:

```
LOAD SAMPLE1 ARIRVSTC
START SAMPLE1 parm1 parm2
```

Single User Mode

Single user mode application programs are programs that run in the same machine as the SQL/DS code and that are under the control of SQL/DS. (In this case, the user machine and the data base machine are actually the same machine.)

Single user mode programs are invoked by starting SQL/DS via the SQLSTART EXEC, provided by IBM. (Before invoking SQL/DS, you must issue IPL CMS.) You must specify both the mode (SYSMODE=S) and your program name (PROGNAME=name) when you issue the SQLSTART EXEC.

When SQLSTART is invoked, SQL/DS loads the program (identified by the PROGNAME parameter) and passes control to it once SQL/DS is initialized. For single user mode, only the module needs to be available.

The *SQL/Data System Operation for VM/SP* manual lists all the initialization parameters you can specify when you start SQL/DS in single user mode. However, a system programmer might determine what the best initialization parameters for your system are, and pass these on to you.

Following is an example of the SQLSTART EXEC for invoking programs in single user mode with no user parameters:

```
SQLSTART DB (SQLDBA) PARM (SYSMODE=S, LOGMODE=A, PROGNAME=SAMPLE1)
```

If your program or SQL/DS ends abnormally, you may receive a “minidump” (depending on what initialization parameters were specified). “Mini-dumps” are described in the *SQL/Data System Planning and Administration for VM/SP* manual.

Specifying User Parameters

When starting SQL/DS in single user mode, you can also specify parameters to be passed to your application program. You should use the PARM keyword of the SQLSTART EXEC for parameter input. The SQLSTART EXEC purges the CMS program and console stacks. Therefore, any program run in single user mode cannot rely on console or program stack input.

You must place a slash (/) between the SQL/DS parameters and the application program parameters, as shown below:

```
SQLSTART DB (SQLDBA) PARM (SYSMODE=S, LOGMODE=A, PROGNAME=SAMPLE1/parm1, parm2)
```

Note: CMS reads only the first 130 characters of the command line. If you must specify many SQL/DS initialization parameters and user parameters, they will not fit on the command line. Therefore, you must use a CMS file for some of the parameters. Remember that SQL/DS does not permit you to specify user parameters in a CMS file. Thus, you should specify the SQL/DS initialization parameters in the CMS file, and the user parameters on the command line.

Further information and examples of installing VM/SP applications can be found in the *SQL/Data System Installation for VM/SP* manual.

Testing and Debugging Concerns

Contents

Error Handling	202
WHENEVER	206
Monitoring Execution Performance	209

Error Handling

As mentioned previously, SQL/DS returns a result code in the SQLCA after executing almost every SQL statement. The only statements that do not return SQLCODEs are SQL declarative statements. (Declarative statements aren't executed; therefore, no SQLCODE can be returned.) BEGIN and END DECLARE SECTION, INCLUDE SQLCA, INCLUDE SQLDA, DECLARE CURSOR, and WHENEVER statements are all declarative. INCLUDE SQLDA is described under "Dynamically Defined Statements" on page 147; WHENEVER is discussed after this section. Never test for an SQLCODE after a declarative statement.

Figure 24 is a representation of the SQLCA structure with host-language independent data type descriptions. (Refer to the appendixes for the SQLCA data types of a particular programming language.)

```
SQLCA -- a structure composed of:
  SQLCAID -- character string of length 8
  SQLCABC -- 31-bit binary integer
  SQLCODE -- 31-bit binary integer
  SQLERRM -- varying character string of maximum length 70
  SQLERRP -- character string of length 8
  SQLERRD -- an array composed of:
    SQLERRD(1) -- 31-bit binary integer
    SQLERRD(2) -- 31-bit binary integer
    SQLERRD(3) -- 31-bit binary integer
    SQLERRD(4) -- 4-byte floating point number
    SQLERRD(5) -- 31-bit binary integer
    SQLERRD(6) -- 31-bit binary integer
  SQLWARN -- a sub-structure composed of:
    SQLWARN0 -- single character
    SQLWARN1 -- single character
    SQLWARN2 -- single character
    SQLWARN3 -- single character
    SQLWARN4 -- single character
    SQLWARN5 -- single character
    SQLWARN6 -- single character
    SQLWARN7 -- single character
    SQLWARN8 -- single character
    SQLWARN9 -- single character
    SQLWARNA -- single character
  SQLEXT -- character string of length 5
```

Note: In FORTRAN, the SQLCA structure is different. See Figure 48 on page 464 for additional information.

Figure 24. SQLCA Structure (in Pseudo Code)

The meanings of the various fields in the SQLCA are as follows:

SQLCAID SQL/DS sets this 8-byte field to 'SQLCAID' when your program first uses the structure. The SQLCAID field is an eye-catcher for programmers when a dump is used for problem determination.

SQLCABC Length of SQLCA, set by SQL/DS when your program first uses the structure.

SQLCODE Summarizes the result of executing the statement. In general, zero denotes successful execution. Codes greater than zero denote normal conditions experienced while executing the statement, such as an end of file or some specific warning conditions. Negative codes represent various abnormal conditions, which may have been caused by either an error in your program or a system failure. You should not continue if a severe error occurs. (These severe errors are documented in the *SQL/Data System Messages and Codes for VM/SP* manual.)

SQLERRM May contain one or more character strings, separated by a X'FF' (a hexadecimal character of all '1' bits). SQL/DS uses these character strings internally when generating messages for its own functions. (For example, SQL/DS uses SQLERRM when generating SQL messages that result from executing the preprocessor or the Data Base Services utility.) Note that applications receive SQLCODEs, not messages.

The message text associated with a particular SQLCODE can be found in the *SQL/Data System Messages and Codes for VM/SP* manual. These message texts, however, often have variables in them (for example, &1, &2, and &3). For these cases, the SQLERRM tokens go in the SQLCODE descriptive text. Thus, in addition to the SQLCODE, you should also print the contents of SQLERRM if you are handling SQL errors in a common routine.

The first two bytes of SQLERRM contain the total length of the string (remember that SQLERRM is varying-length).

SQLERRP If the SQLCODE is negative, SQLERRP contains the name of the SQL/DS routine that discovered the error. Together with SQLERRD, this information may be helpful in diagnosing failures.

SQLERRD A collection of six variables that describe the current internal state of SQL/DS. This information is helpful in diagnosing SQL/DS failures or processing status. Only variables 1-4 are used by SQL/DS; the last two are reserved:

1. Relational Data System (RDS) or Resource Manager return code. (RDS and the Resource Manager are internal components of SQL/DS.)
2. Data Base Storage System (DBSS) or Resource Manager return code. (DBSS and the Resource Manager are internal components of SQL/DS.)

3. Number of rows processed, where applicable.
4. SQL/DS cost estimate: A relative value incorporating I/O requirements with a weighted factor of processor requirements for a query. When preparing a dynamically defined SQL statement, you can use this field to determine expected relative performance of the prepared SQL statement.

SQLWARN Characters that warn of various conditions encountered during the processing of your statement. Alternatively, specific warnings may be indicated by positive values in the SQLCA field, SQLCODE. For example, a warning indicator is set when SQL/DS ignores null values in computing an average. When SQL/DS encounters a particular condition, it sets the corresponding warning character to 'W'; otherwise it sets the character to blank. One or more warning characters may be set to 'W' regardless of the code returned in SQLCODE. The meanings of the warning characters are:

SQLWARN0 Set to 'W' if one or more other warning characters is equal to 'W'. Provides a quick test for the existence of any warning. Set to 'S' if SQLWARN6 is set to 'S'.

SQLWARN1 One or more of the requested data items was truncated because of insufficient space in the host variable you provided for output. This flag is set only for character data items; SQL/DS truncates certain numeric data items without setting a warning flag or returning a negative SQLCODE. (See "Data Conversion" on page 76 for more information.) The data items that were truncated can be found by examining the null indicator variables of the data items returned. A positive value in the null indicator denotes the actual length of the variable before truncation.

SQLWARN2 Null values were ignored in the computation of a built-in function (AVG, SUM, MAX, or MIN). This flag is set only during preprocessing, never at run-time.

SQLWARN3 The number of items in the SELECT list is not equal to the number of target variables in the INTO clause. The number of items returned is the minimum of these two numbers.

SQLWARN4 An UPDATE or DELETE statement has been used without a WHERE clause. You should verify that the update or deletion was intended unconditionally on the entire table.

This flag is set only during preprocessing, never at run-time.

SQLWARN5 A WHERE clause, represented internally by one or more search arguments, associated with a SELECT statement has exceeded an SQL/DS internal limitation. This means that a performance degradation will result because SQL/DS will not internally convert eligible predicates to search arguments. SQL/DS may still choose to use indexes for eligible predicates, but if not, degradation may further be increased since a DBSPACE scan would be used to retrieve query data.

Decreasing the number of predicates, such as by removing unnecessary conditions which may exist in the WHERE clause of the SELECT statement, may alleviate this condition.

SQLWARN6 Set to 'W' if the last SQL statement executed caused SQL/DS to terminate a logical unit of work. This flag is *not* set after a ROLLBACK WORK statement, but is set by SQL/DS when the rollback of the logical unit of work is implicit. For example, SQLWARN6 is set when a logical unit of work is backed out due to a deadlock. (Deadlocks are explained under "SQL/DS Automatic Locking Mechanism" on page 232.)

Set to 'S' when SQL/DS issues an SQLCODE which is considered "severe." The list of severe SQLCODEs includes:

-805	-932	-938
-806	-933	-940
-807	-934	-941
-902	-935	
-931	-937	

Severe errors are those that place SQL/DS in an unusable state. Therefore, any further attempts by the application to access SQL/DS will cause the application to be abnormally terminated.

SQLWARN7 Reserved for SQL/DS use.

SQLWARN8 A statement has been disqualified for blocking for reasons other than storage. For example, SQLWARN8 is set if long fields or a FOR UPDATE clause were used in a statement. (See "To Block or Not to Block?" on page 254 for more information on blocking.)

SQLWARN9 Blocking was cancelled for a cursor because of insufficient storage in the user virtual machine.

SQLWARNA Blocking was cancelled for a cursor because a blocking factor of at least two rows could not be maintained.

SQLEXT Reserved for SQL/DS use.

Because there is only one return code structure in each program, you should copy out of the structure any information that you wish to save before the next SQL statement is executed. Of particular note are the SQLCODE and the warning indicators (SQLWARN). An SQL statement (called WHENEVER) allows you to detect abnormal conditions and take appropriate action.

WHENEVER

Formats:

```
WHENEVER SQLERROR {STOP | CONTINUE | {GO TO|GOTO} statement-label}  
WHENEVER SQLWARNING {STOP | CONTINUE | {GO TO|GOTO} statement-label}  
WHENEVER NOT FOUND {CONTINUE | {GO TO|GOTO} statement-label}
```

Note: The STOP condition is not valid for FORTRAN applications.

Examples:

```
WHENEVER SQLERROR GOTO ERRORX  
WHENEVER SQLWARNING CONTINUE  
WHENEVER NOT FOUND CONTINUE
```

Authorization:

Anyone connected to SQL/DS can issue this statement.

The WHENEVER statement lets you specify an action to be taken depending on what SQL/DS returns in the SQLCA. The WHENEVER statement is declarative; it is not executed at run-time and returns no SQLCODE.

The keywords SQLERROR, SQLWARNING, and NOT FOUND in the statement syntax above identify some SQLCA condition. The SQLERROR condition exists when SQL/DS has set SQLCODE to a negative value. The SQLWARNING condition exists when SQL/DS sets SQLWARN0 to 'W'. The NOT FOUND condition exists when SQLCODE is set to 100.

The braced keywords define the action to be taken whenever the specified SQLCA condition occurs:

- STOP** causes program termination. If a logical unit of work is in progress, it is rolled back. Note that you can't specify STOP for WHENEVER NOT FOUND or in FORTRAN applications.
- CONTINUE** causes the next sequential program instruction to be executed. (The SQLCA condition is ignored.) If a fatal error is encountered, however, the program should go no further. Fatal errors are documented in the *SQL/Data System Messages and Codes for VM/SP* manual.
- GO TO (or GOTO)** causes control to pass to the statement at the specified label. The statement label cannot exceed 18 characters unless the host language has additional limitations.

If you don't write a WHENEVER statement, SQL/DS acts as if you had coded the following statements in your program:

```
WHENEVER SQLERROR CONTINUE
WHENEVER SQLWARNING CONTINUE
WHENEVER NOT FOUND CONTINUE
```

The scope of a WHENEVER statement is determined by its position in the source program listing, not by its placement in the logic flow. (This is because WHENEVER is a declarative statement.) For example:

```
DO WHILE (X > Y)
  EXEC SQL CREATE INDEX I1 ON SUPPLIERS (NAME)
  .
  (host language code)
  .
  .
  EXEC SQL DELETE FROM QUOTATIONS
    WHERE PRICE > 2000
  EXEC SQL WHENEVER SQLERROR STOP
  .
  (host language code)
  .
  .
  EXEC SQL SELECT SUPPNO, NAME FROM SUPPLIERS ...
  .
  (host language code)
  .
  .
  EXEC SQL WHENEVER SQLERROR CONTINUE
END-DO
EXEC SQL DROP INDEX I1
```

In the pseudo code program fragment above, the scope of the first WHENEVER is only the SELECT statement. The second WHENEVER applies to the DROP INDEX statement (and to all SQL statements that follow it -- until another WHENEVER is encountered). The CREATE INDEX and DELETE statements are not covered by a WHENEVER (there is no preceding WHENEVER); therefore, the default SQLERROR CONTINUE action applies. Note that the scope of the WHENEVER is independent of the *execution sequence* of statements.

You can test the elements of the SQLCA for general or specific warning or error conditions in addition to or instead of using the WHENEVER statement. To do

this, use a **WHENEVER** statement with a **CONTINUE** or **GOTO** somewhere in the source program before the **SQL** statements for which you want to directly examine the **SQLCA**. For example, Figure 25 shows pseudo code for a typical error handling routine:

```
EXEC SQL WHENEVER SQLERROR GOTO LABX
      .
      .
      .
LABX   SAVE = SQLCODE
      EXEC SQL WHENEVER SQLERROR CONTINUE
      EXEC SQL ROLLBACK WORK RELEASE
      DISPLAY ('PROGRAM TERMINATED. SQLCODE IS:')
      DISPLAY (SAVE)
      STOP
```

Figure 25. Pseudo-Code Error Handling Routine

On an **SQLERROR** condition, control passes to **LABX**. The **SQLCODE** is immediately saved because **SQLCODE** is changed when the **ROLLBACK WORK** is executed. (**WHENEVER** statements never return an **SQLCODE**.) The **WHENEVER SQLERROR CONTINUE** statement prevents a program loop resulting from **ROLLBACK WORK** producing an error. Once the logical unit of work is rolled back, informational messages are displayed and the program terminates.

As noted earlier, **SQL** declarative statements never return an **SQLCODE**. You should never test for an **SQLCODE** after these statements. You should, however, examine the **SQLCA** for all data manipulation statements. **INSERT** and **UPDATE** statements, in particular, can fail after processing some rows of a table. In that case, you would usually issue a **ROLLBACK WORK** before you terminate your program so the data base isn't left in a partially updated state. If you omit the **WHERE** clause in a **DELETE** statement, **SQL/DS** sets **SQLWARN4** on so that you have a chance to verify that all the rows of the table are to be deleted.

Another reason an application might want to process the **SQLERROR** condition is for graceful cleanup and termination. An example of this is the **SQL/DS ISQL** transaction. Rather than terminating the **ISQL** session, the user is given an error message and allowed to proceed. In fact, in some cases, the terminal user is given the opportunity to indicate whether backout is necessary or not.

Refer to the *SQL/Data System Messages and Codes for VM/SP* manual for all **SQLCODE** descriptions.

Monitoring Execution Performance

You can use an EXPLAIN command to retrieve information about the structure and execution performance of SQL commands. The EXPLAIN command can be coded into COBOL, PL/I, Assembler, and FORTRAN programs. (It can also be issued via the DBS utility or ISQL.)

The EXPLAIN command accepts as an argument another SQL command. When executed, the EXPLAIN command analyzes the performance and structure of an SQL command and places the information into one or more SQL/DS explanation tables. This information can be used to:

- Analyze request loads
- Estimate the size of responses
- Separate queries into their subquery structures
- Obtain costs for statements
- Assist in data base design
- Determine when a program needs to be preprocessed again.

The format for the EXPLAIN command is as follows:

```
EXPLAIN explain-spec [SET QUERYNO = small-integer-value] FOR sql-command
```

explain-spec

is the name of the explanation table(s) into which information is to be placed. explain-spec may include one or more of the following options, separated by commas:

REFERENCE for information contained in the REFERENCE__TABLE

STRUCTURE for information contained in the STRUCTURE__TABLE

COST for information contained in the COST__TABLE

PLAN for information contained in the PLAN__TABLE

ALL for information contained in all of the above tables.

small-integer-value

is an integer constant that can fit into a SMALLINT field. The SET QUERYNO clause allows you to place an integer value into the QUERYNO fields of the rows in the explanation tables. Assigning a different number on each EXPLAIN will make it easier to identify information collected.

sql-command

is the SQL command to be analyzed. You can analyze UPDATE, DELETE, and INSERT commands as well as SELECT commands. (SELECT commands are considered the primary candidates for EXPLAIN analysis). sql-command is not a quoted string and must not be put in a host variable.

The length of the SQL statement is limited to about 8000 characters.

To use the EXPLAIN command, you must own an explanation table for each of the specified explain-spec options. For example, if you use the COST and STRUCTURE options, you must own a COST__TABLE and a STRUCTURE__TABLE. If you don't own the needed tables, EXPLAIN has no effect, and returns an error code.

The result tables built by the EXPLAIN command are created during preprocessing of the containing program. After preprocessing, execution of an EXPLAIN command has no meaning and the results are unpredictable. Nevertheless, you can PREPARE/EXECUTE or EXECUTE IMMEDIATE an EXPLAIN command.

For additional information on using the EXPLAIN command, refer to the *SQL/Data System Planning and Administration for VM/SP* manual.

Putting the Program into Production

Contents

Authorization	213
When Does a Creator Get the RUN Privilege?	213
When Does a Creator Get the GRANT Option?	214
How SQL/DS Uses the Catalogs for Program Authorization	215
How SQL/DS Implements Program Authorization	216
Decision Tables Used to Determine Program Authorization	216
ACQUIRE DBSPACE	217
ALTER DBSPACE	217
ALTER TABLE	218
COMMENT	218
CREATE INDEX	219
CREATE TABLE	219
DELETE	220
INSERT	221
SELECT	222
UPDATE	224
LOCK DBSPACE	225
LOCK TABLE	226
Data Control	226
Acquiring a DBSPACE	227
Dropping a DBSPACE	230
Changing DBSPACE Characteristics	231
SQL/DS Automatic Locking Mechanism	232
Ending the Logical Unit of Work	232
COMMIT WORK	233
ROLLBACK WORK	234
Using the LOCK Statement to Override Automatic Locking	235
Use of Long Fields	236
Updating Internal Statistics	237
Data Definition	237
Creating a Table	238
Adding a Column to a Table	241
Dropping a Table	241
Creating an Index	242
Dropping an Index	245
Creating a Synonym	245
Dropping a Synonym	246
Putting Comments into SQL/DS Catalogs	247

Putting Labels on Tables or Columns	249
Performance Considerations	250
Selecting the Isolation Level	251
To Block or Not to Block?	254
Including External Source Files	255
Including Secondary Input	255




Authorization

As you know, all SQL/DS programs must be preprocessed before they are compiled or assembled. As a result of a preprocessor run, SQL/DS creates an access module. SQL/DS uses the access module to satisfy data base requests at run time. When a program is preprocessed, SQL/DS also determines who has the capability to run the access module. SQL/DS bases this determination on the type of access requested. That is, SQL/DS considers the statement used, the ownership of the accessed objects, and other factors when determining who can run an access module, and who can't. This section describes this determination process.

When Does a Creator Get the RUN Privilege?

At the completion of the preprocessor run, SQL/DS tells the creator whether or not an access module was generated. SQL/DS generates an access module and gives the creator the RUN privilege for it if:

1. All referenced objects exist when the program is preprocessed; and,
2. The creator has explicit authorization for all SQL statements used in the access module.



There are other cases where the RUN privilege is possible. In these cases, however, successful execution depends on the existence of the objects and the authority of the creator **at run time**. This is because SQL/DS marks certain statements for re-checking at run time if the above requirements for the RUN privilege are not met when the program is preprocessed. Here are a few specific examples of operations that allow the RUN privilege (with a subsequent re-check):

- A non-existing table (at preprocess time) is referenced, and an explicit qualifier is used to indicate the creator of the table. The creator specified is not the creator of the program.
- A creator with DBA authority creates an index on a non-existing (at preprocess time) table.
- A creator with DBA authority codes INSERT, DELETE, UPDATE, or SELECT statements that refer to a non-existing (at preprocess time) table.
- A creator with DBA authority refers to a table in a WHERE clause. The creator, however, lacks explicit authority.

How SQL/DS determines whether to grant the RUN privilege in such cases is not covered by a single formula. Rather, decision tables are used for each SQL statement. The decision tables are presented in a later section.

When Does a Creator Get the GRANT Option?

In addition to the RUN privilege, SQL/DS may also give a creator the GRANT option. The GRANT option is also a privilege of sorts. The GRANT option allows the creator to grant the RUN privilege of the access module to other users. In following discussions, the RUN privilege with the GRANT option is called the *GRANT RUN* privilege.

As with RUN authority, SQL/DS uses decision tables to determine when a creator receives the GRANT RUN privilege. These decision tables are in a later section.

Following are some of the circumstances that allow a creator to gain the GRANT RUN privilege:

- SQL statements in the program require RESOURCE authority, and the creator has RESOURCE authority.
- All objects that are referred to in the program exist when the program is preprocessed.
- The creator has the necessary authority (with the GRANT option) to access any referenced object that the creator does not own.
- The preprocessor run did not result in any error diagnostics.
- No statements required DBA authority. The following are examples of operations that require DBA authority:
 - Acquiring a PUBLIC DBSPACE
 - Creating a table in another user's DBSPACE or in a SYSTEM DBSPACE
 - Acquiring a DBSPACE for another user
 - Altering another user's table when the creator doesn't have explicit ALTER authority on the table
 - Locking another user's DBSPACE
 - Commenting on another user's table
 - Dropping another user's object
 - Locking another user's table
 - Altering another user's DBSPACE
 - Creating an index on another user's table when the creator doesn't have explicit INDEX authority on that table
 - Creating a table for another user

- Inserting, deleting, or updating another user's table when the creator doesn't have the explicit authority to do so.

Note: The following statements also require DBA authority. These statements, however, do not affect the RUN privilege, because they are not checked until execution time:

- ALTER DBSPACE when the creator qualifier is not given
- LOCK DBSPACE when the creator qualifier is not given
- DROP DBSPACE when the creator qualifier is not given
- CREATE TABLE in someone else's DBSPACE or in a SYSTEM DBSPACE when the DBSPACE creator qualifier is not given.

How SQL/DS Uses the Catalogs for Program Authorization

SQL/DS records the current RUN and GRANT RUN privileges held by all users in the SYSPROGAUTH SQL/DS catalog. The entries in the catalog identify:

- The grantor
- The grantee
- The access module that is the subject of the RUN privilege
- A marker indicating that the grantee holds either RUN ('Y') or GRANT RUN ('G') authority.

The entries are added to the catalog when a preprocessor run completes. The entries made depend, of course, on whether the program satisfies the various conditions described in the preceding sections. SQL/DS also makes entries in the SYSPROGAUTH catalog when a user grants the RUN privilege to another user.

SQL/DS also updates the SYSUSERAUTH, SYSCOLAUTH, and SYSTABAUTH catalogs. In these catalogs, SQL/DS records the program's dependency on some authorization. For example, when a program requires RESOURCE authority to execute successfully, SQL/DS makes an entry in SYSUSERAUTH to reflect that dependency. The catalog entries help SQL/DS keep track of which access modules are valid, and which are invalid.

It should be noted that SYSTEM is the owner of all catalog tables; you must qualify all catalog tables with that name, unless you have a synonym defined. All SQL/DS catalogs are described in the *SQL/Data System Planning and Administration for VM/SP* manual.

How SQL/DS Implements Program Authorization

As noted earlier, SQL/DS uses decision tables to determine whether the creator is authorized to execute a given statement. At preprocess time, SQL/DS evaluates each statement and assigns it an authorization “score.” At the end of the preprocessor run, SQL/DS picks the lowest “score” of all the statements. SQL/DS uses the low score as the program’s authorization. There are three possible scores for each statement:

- ‘G’** means that the creator has the necessary authorization for this statement such that the creator can receive the GRANT RUN privilege.
- ‘Y’** means that the creator has the necessary authorization for this statement such that the creator can receive the RUN privilege, but not the GRANT option.
- ‘D’** means that the creator must have DBA authority to execute the program containing this statement. No entry is made in the authorization catalogs.

‘G’ is the highest score, followed by ‘Y’, followed by ‘D’. For example, suppose a program contains three statements. On two of the statements, the creator receives a ‘G’, but on the third statement, the creator receives a ‘Y’. In this case, SQL/DS assigns the program a ‘Y’ (the lowest score). The ‘Y’ means that the creator can run the program, but cannot grant the RUN privilege on the program to another user.

‘G’, ‘Y’, and ‘D’ are used in the following decision tables. In addition, “N/A” is used. “N/A” stands for “Not Applicable.”

Some entries in the decision tables indicate a letter value, but also indicate that a negative SQLCODE will be returned. In these cases, the statement still receives the letter “score,” but the negative SQLCODE means that SQL/DS will re-check the statement at execution time.

Note: -204 SQLCODE results in an ARI587I message during a preprocessor run.

Decision Tables Used to Determine Program Authorization

This section contains the decision tables that SQL/DS uses to determine whether to grant the RUN privilege. A decision table is presented for each applicable SQL statement.

When reading the following charts, remember that the *creator* (or *author*) is the user who preprocessed the program.

ACQUIRE DBSPACE

		DBSPACE Owner		
		1	2	3
Authority of Creator	V	PUBLIC	PRIVATE	
			Owner is Creator	Owner is not Creator
A	DBA	D	G	D
B	RESOURCE	G and -551 SQLCODE	G	G and -551 SQLCODE
C	None of the Above	G and -551 SQLCODE	G and -552 SQLCODE	G and -551 SQLCODE

For cases A2 and B2, SQL/DS makes an entry in the SYSUSERAUTH catalog with RESOURCE set to 'Y'. The entry indicates the program's dependency.

ALTER DBSPACE

		DBSPACE Owner		
		1	2	3
Authority of Creator:		No Owner Specified	Creator is Owner	Creator is not Owner
A	DBA	G	G	D
B	Non-DBA	G	G	G and -551 SQLCODE

ALTER TABLE

		Owner of Table			
		1	2	3	4
Authority of Creator:		SYSTEM	Owner is NOT the creator of this program	Owner is the creator of this program	Table does not exist
A	DBA and no ALTER authority	D	D	N/A	D
B	ALTER authority	N/A	Y	N/A	N/A
C	ALTER authority with GRANT option.	N/A	G	G	N/A
D	No DBA and no ALTER authority	G and -552 SQLCODE	G and -551 SQLCODE	N/A	G and -551 SQLCODE

For cases B2, C2, and C3, SQL/DS makes entries in the SYSTABAUTH catalog with the ALTERAUTH columns set to 'Y'. The entries represent this program's dependency on ALTER authority.

COMMENT

		Object Owner	
		1	2
Authority of Creator:		Creator is not Owner	Creator is Owner
A	DBA	D	G
B	Non-DBA	G and -551 SQLCODE	G

CREATE INDEX

		1	2	3	4
		Table Exists		Table Not Created	
Authority of Creator:		Creator is Owner	Creator is not Owner	Creator is Owner	Creator is not Owner
A	DBA, no INDEX authority	N/A	D	G and -204 SQLCODE	Y and -204 SQLCODE
B	Non-DBA, INDEX authority with GRANT option	G	G	N/A	N/A
C	Non-DBA, INDEX authority	N/A	Y	N/A	N/A
D	Non-DBA, no INDEX authority	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE

For cases B1, B2, and B3, SQL/DS makes an entry in the SYSTABAUTH catalog with the INDEXAUTH column set to 'Y'.

Note that it is possible for an owner of a table to create an index on that table in the name of another user. This is true even if the table owner does not have DBA authority.

CREATE TABLE

		1	2	3
		Creator is SYSTEM	Creator is Owner	Creator is not Owner
A	DBA	-550 SQLCODE	G	D
B	Non-DBA	-550 SQLCODE	G	G and -551 SQLCODE

In cases A1 and B1 SQL/DS no access module is created.

DELETE

There are two decision tables that apply to DELETE:

Table from Which DELETE Is Made

Table to which DELETE is applied		1	2	3	4	5	6
		Owner is System		Table Exists		Table not Created	
Authority of Author:		TNAME is SYS-CATALOG	TNAME is not SYS-CATALOG	Author is Owner	Author is not Owner	Author is Owner	Author is not Owner
A	DBA, no DELETE authority	D	G and -823 SQLCODE	N/A	D	G and -204 SQLCODE	Y and -204 SQLCODE
B	Non-DBA, DELETE with GRANT Option	N/A	G and -823 SQLCODE	G	G	N/A	N/A
C	Non-DBA, DELETE authority	N/A	G and -823 SQLCODE	N/A	Y	N/A	N/A
D	Non-DBA, no DELETE authority	G and -552 SQLCODE	G and -823 SQLCODE	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE

In cases B3, B4, and C4, SQL/DS makes entries in the SYSTABAUTH catalog. The entries have the DELETAUTH column set to 'Y' to indicate the program's dependency.

For cases A1, A2, A3, and A4, the SYSPROGAUTH entry is made, but the -823 SQLCODE indicates that the statement will *always* fail on subsequent re-checks during execution.

Tables Referenced in the WHERE Clause: Note that the authorization checking in the previous decision table precedes the logic of this table. If the first decision table yields a negative SQLCODE, processing stops. Otherwise, SQL/DS applies the lowest level of authorization gained from the two decision tables.

Authority of Author:		1		2		3		4	
		Table Exists				Table Not Created			
		Author is Owner	Author is not Owner	Author is Owner	Author is not Owner	Author is Owner	Author is not Owner	Author is Owner	Author is not Owner
A	DBA, no SELECT authority	N/A	Y	G and -204 SQLCODE	Y and -204 SQLCODE				
B	Non-DBA, SELECT authority with GRANT option	G	G	N/A	N/A				
C	Non-DBA, SELECT authority	N/A	Y	N/A	N/A				
D	Non-DBA, no SELECT authority	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE				

In cases B1, B2, and C2, SQL/DS makes entries in the SYSTABAUTH catalog. These entries have the SELECTAUTH column set to 'Y' to indicate the program's dependency.

In case A2, SQL/DS makes an entry in the SYSUSERAUTH catalog to show the program's dependency on DBA authority.

INSERT

There are two decision tables that apply to INSERT:

Table into Which the INSERT Is Made

Authority of Author:		1		2		3		4		5	
		Table Exists				Table Not Created					
		Owner is SYSTEM	Author is Owner	Author is not Owner	Author is Owner	Author is not Owner	Author is Owner	Author is not Owner	Author is Owner	Author is not Owner	
A	DBA, no INSERT authority	D	N/A	D	G and -204 SQLCODE	Y and -204 SQLCODE					
B	Non-DBA, INSERT authority with GRANT Option	G and -552 SQLCODE	G	G	N/A	N/A					
C	Non-DBA, INSERT authority	G and -552 SQLCODE	N/A	Y	N/A	N/A					
D	Non-DBA, no INSERT authority	G and -552 SQLCODE	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE					

In cases B2, B3, and C3, SQL/DS makes entries in the SYSTABAUTH catalog. The entries have the INSERTAUTH column set to 'Y' to indicate the program's dependencies.

Tables Referenced in the WHERE Clause: Note that the authorization checking in the previous decision table precedes the logic of this table. If the first decision table yields a negative SQLCODE, processing stops. Otherwise, SQL/DS applies the lowest level of authorization gained from the two decision tables.

		1	2	3	4
		Table Exists		Table Not Created	
Table in WHERE clause		Author is Owner	Author is not Owner	Author is Owner	Author is not Owner
A	DBA, no SELECT authority	N/A	Y	G and -204 SQLCODE	Y and -204 SQLCODE
B	Non-DBA, SELECT authority with GRANT option	G	G	N/A	N/A
C	Non-DBA, SELECT authority	N/A	Y	N/A	N/A
D	Non-DBA, no SELECT authority	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE

In cases B1, B2, and C2, SQL/DS makes entries in the SYSTABAUTH catalog. These entries have the SELECTAUTH column set to 'Y' to indicate the program's dependency.

In case A2, SQL/DS makes an entry in the SYSUSERAUTH catalog to show the program's dependency on DBA authority.

SELECT

There are two decision tables that apply to SELECT:

Tables in the FROM List

		1	2	3	4
		Table Exists		Table Not Created	
Authority of Author:		Author is Owner	Author is not Owner	Author is Owner	Author is not Owner
A	DBA, no SELECT authority	N/A	Y	G and -204 SQLCODE	Y and -204 SQLCODE
B	Non-DBA, SELECT authority with GRANT option	G	G	N/A	N/A
C	Non-DBA, SELECT authority	N/A	Y	N/A	N/A
D	Non-DBA, no SELECT authority	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE

In cases B1, B2, and C2, SQL/DS makes entries in the SYSTABAUTH catalog. The entries have the SELECTAUTH column set to 'Y' to indicate the program's dependency.

In case A2, there are some instances where a 'Y' entry is made in the DBAAUTH field of the SYSUSERAUTH catalog, showing program dependencies on DBA authority.

Tables Referenced in the WHERE Clause: Note that the authorization checking in the previous decision table precedes the logic of this table. If the first decision table yields a negative SQLCODE, processing stops. Otherwise, SQL/DS applies the lowest level of authorization gained from the two decision tables.

		1	2	3	4
		Table Exists		Table Not Created	
Authority of Author:		Author is Owner	Author is not Owner	Author is Owner	Author is not Owner
A	DBA, no SELECT authority	N/A	Y	G and -204 SQLCODE	Y and -204 SQLCODE
B	Non-DBA, SELECT authority with GRANT option	G	G	N/A	N/A
C	Non-DBA, SELECT authority	N/A	Y	N/A	N/A
D	Non-DBA, no SELECT authority	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE

In cases B1, B2, and C2, SQL/DS makes entries in the SYSTABAUTH catalog. These entries have the SELECTAUTH column set to 'Y' to indicate the program's dependency.

In case A2, SQL/DS makes an entry in the SYSUSERAUTH catalog to show the program's dependency on DBA authority.

UPDATE

There are two decision tables that apply to UPDATE:

Tables on Which the Update Is Made

		1	2	3	4
		Table Exists		Table Not Created	
Table on which UPDATE is made		Author is Owner	Author is not Owner	Author is Owner	Author is not Owner
Authority of Author:					
A	DBA, no UPDATE authority	N/A	D	G and -204 SQLCODE	Y and -204 SQLCODE
B	Non-DBA, UPDATE authority with GRANT option	G	G	N/A	N/A
C	Non-DBA, UPDATE authority	N/A	Y	N/A	N/A
D	Non-DBA, no UPDATE authority	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE

In cases B1, B2, and C2, SQL/DS makes entries in the SYSCOLAUTH catalog and the SYSTABAUTH catalog. These entries show that the program depends on UPDATE authority (the UPDATEAUTH column) for specific columns. SQL/DS makes the entries without setting the GRANT option to 'Y'.

Tables Referenced in the WHERE Clause: Note that the authorization checking in the previous decision table precedes the logic of this table. If the first decision table yields a negative SQLCODE, processing stops. Otherwise, SQL/DS applies the lowest level of authorization gained from the two decision tables.

		1	2	3	4
		Table Exists		Table Not Created	
Authority of Author:		Author is Owner	Author is not Owner	Author is Owner	Author is not Owner
A	DBA, no SELECT authority	N/A	Y	G and -204 SQLCODE	Y and -204 SQLCODE
B	Non-DBA, SELECT authority with GRANT option	G	G	N/A	N/A
C	Non-DBA, SELECT authority	N/A	Y	N/A	N/A
D	Non-DBA, no SELECT authority	N/A	G and -551 SQLCODE	G and -204 SQLCODE	Y and -204 SQLCODE

In cases B1, B2, and C2, SQL/DS makes entries in the SYSTABAUTH catalog. These entries have the SELECTAUTH column set to 'Y' to indicate the program's dependency.

In case A2, SQL/DS makes an entry in the SYSUSERAUTH catalog to show the program's dependency on DBA authority.

LOCK DBSPACE

		1	2
Authority of Author:		Author is Owner	Author is not Owner
A	DBA	G	D
B	Not DBA	G	G and -551 SQLCODE

LOCK TABLE

		Owner of Table		
Authority of Author:		1	2	3
		Owner is Author	Owner is not Author	Table does not Exist
A	DBA, and no SELECT authority	N/A	D	D
B	Non-DBA, SELECT authority with GRANT option	G	G	N/A
C	Non-DBA, SELECT authority	N/A	Y	N/A
D	Non-DBA, no SELECT authority	N/A	G and -551 SQLCODE	G and -551 SQLCODE

For cases B1, B2, and C2, SQL/DS makes entries in the SYSTABAUTH catalog. The entries have the SELECTAUTH column set to 'Y' to show the program's dependency.

Data Control

SQL Data Control statements manage logical units of work and *DBSPACEs*, which are units of space. More specifically, with data control statements you can:

- Acquire and drop *DBSPACEs* (ACQUIRE *DBSPACE* and DROP *DBSPACE*)
- Change *DBSPACE* characteristics (ALTER *DBSPACE*)
- End a logical unit of work and either commit or rollback the changes you made (COMMIT WORK and ROLLBACK WORK)
- Override the SQL/DS automatic locking mechanism (LOCK)
- Update internal statistics (UPDATE STATISTICS).

Thus, data control statements affect the areas in which your tables and program access modules reside. They also affect the way in which programs work with the data base.

Acquiring a DBSPACE

Format:

```
ACQUIRE {PUBLIC | PRIVATE} DBSPACE  
        NAMED [owner.]dbspace-name
```

```
        ( [ NHEADER = {8|integer}  
          PAGES    = {128|integer}  
          PCTINDEX = {33|integer}  
          PCTFREE  = {15|integer}  
          LOCK     = {PAGE|size}  
          STORPOOL = {integer} ] )
```

The LOCK parameter is applicable to PUBLIC DBSPACES only.

Example:

```
ACQUIRE PRIVATE DBSPACE NAMED MFBSPACE  
        (STORPOOL=3, PCTFREE=25)
```

Authorization:

You must have DBA authority to acquire either a PUBLIC DBSPACE or a DBSPACE for another user. You must have RESOURCE authority to acquire a PRIVATE DBSPACE.

The ACQUIRE DBSPACE statement causes SQL/DS to find an available DBSPACE of the requested type (PUBLIC or PRIVATE) and give it the dbspace-name you specify. The dbspace-name must be an SQL identifier, as described in Chapter 1; you can use it to refer to the DBSPACE in other SQL statements, such as CREATE TABLE.

If the DBSPACE type is PUBLIC, its owner becomes PUBLIC; if the type is PRIVATE, its owner becomes the user who preprocessed the program in which the ACQUIRE DBSPACE is embedded. DBSPACE names must be unique within all the DBSPACES owned by the same user, but may duplicate the name of a DBSPACE owned by another user.

If you have DBA authority, you can acquire a DBSPACE for another user by concatenating the userid to the dbspace-name:

```
ACQUIRE PRIVATE DBSPACE NAMED JONES.SPACE1
```

In the above statement, the owner of the DBSPACE is user JONES. User JONES can refer to it as simply SPACE1.

You can optionally specify the following properties of a DBSPACE. If you specify more than one, you can specify them in any order. You must separate the parameters with commas.

NHEADER Number of Header Pages. The number of 4096-byte logical pages in the DBSPACE that SQL/DS reserves for header pages. SQL/DS uses header pages to record information about the contents of the DBSPACE.

Notes:

1. *NHEADER cannot be larger than eight pages.*
2. *If NHEADER is not specified, the default is eight pages.*
3. *You cannot change NHEADER after the DBSPACE has been acquired. If you choose a small number for NHEADER, it may limit the number of different tables that can be created in the DBSPACE.*

PAGES Number of Pages. The minimum number of 4096-byte logical pages you require for this DBSPACE.

Notes:

1. *SQL/DS may actually give you more pages than you request because it acquires storage in units of 128 pages. However, of the available DBSPACES, the one chosen will be the smallest that will satisfy the size specified for PAGES. SQL/DS determines the number of pages you receive by rounding the number you specify to the next higher multiple of 128 pages. For example, if you specify PAGES=53, SQL/DS acquires a block of 128 pages. If you specify PAGES=130, SQL/DS acquires 256 pages.*
2. *If you do not specify PAGES, SQL/DS acquires the smallest available DBSPACE (128 pages) by default.*

PCTINDEX Percentage of Index Pages. The percentage of all pages in the DBSPACE that SQL/DS is to reserve for the construction of indexes.

Notes:

1. *If you don't specify PCTINDEX, the default is 33 percent.*
2. *You cannot change PCTINDEX after the DBSPACE has been acquired. If you choose a small number for PCTINDEX, it may limit the number of indexes that can be created on tables in the DBSPACE. (If you find that the PCTINDEX is too small, you can acquire another DBSPACE and move the data to it.)*

PCTFREE

Percentage of Free Space. The percentage of the space on *each* page that SQL/DS is to keep empty when data is inserted into the DBSPACE.

Notes:

1. *If you don't specify PCTFREE, the default is 15 percent.*
2. *Typically a user might acquire a DBSPACE with PCTFREE set to some value such as 25 percent. The DBSPACE is then loaded with data via the Data Base Services utility (described in the SQL/Data System Data Base Services Utility for VM/SP manual). SQL/DS ensures that at least 25 percent of the space on each page is left empty. After the initial loading of the DBSPACE, the user can set PCTFREE to zero by means of the ALTER DBSPACE statement (described later). Then, in subsequent insertions, SQL/DS is free to place new data in the space reserved during initial loading. Using reserved free space in this way results in a more favorable physical clustering of data on pages when the data is loaded, and, therefore, improves access time. The SQL/Data System Planning and Administration for VM/SP manual discusses data clustering in more detail.*
3. *The value of PCTFREE is critical during mass insertion of data into a DBSPACE (for example, a DBS Utility DATALOAD command). Refer to the appendix "Estimating the Number of Data Pages Required" in SQL/Data System Planning and Administration for VM/SP for more information on the DBSPACE percent free specification.*

LOCK

Lock Size. Applicable to PUBLIC DBSPACES only. (PRIVATE always locks a DBSPACE.) The valid specifications for size are DBSPACE, PAGE, and ROW.

Notes:

1. *The lock size determines the size of the locks that SQL/DS acquires when a user reads or updates data. If you specify ROW, SQL/DS locks only an individual row in the table; PAGE or DBSPACE cause the smallest lockable unit to be a page (4096 bytes) or a DBSPACE, respectively. Key level locking is used for indexes or tables in DBSPACES for which row level locking is specified.*
2. *In general, larger locking units (for example, DBSPACE) cause less overhead to be spent in acquiring locks, but also limit concurrency.*
3. *The default LOCK for each PUBLIC DBSPACE is PAGE.*

STORPOOL Storage Pool Number. This parameter indicates that SQL/DS must acquire this DBSPACE from the specified storage pool.

Notes:

1. *If a DBSPACE of the specified type and size is not available in this storage pool, the ACQUIRE DBSPACE is not successful; SQL/DS returns a negative SQLCODE.*
2. *If you don't specify STORPOOL, SQL/DS acquires a DBSPACE of the correct type and size from any recoverable storage pool. To acquire a DBSPACE from a non-recoverable storage pool, you must specify the STORPOOL parameter.*
3. *You can also define storage pools that are not recoverable. Non-recoverable data reduces system overhead. This is done at the expense of automatic recovery for data update. That is, the burden of recovery is placed on the user. Non-recoverable storage pools are particularly useful in cases where large amounts of data are loaded from an external source, and that data is never modified thereafter. See SQL/Data System Planning and Administration for VM/SP for more information.*

Dropping a DBSPACE

Format:

```
DROP DBSPACE [owner.]dbspace-name
```

Examples:

```
DROP DBSPACE MFBSpace  
DROP DBSPACE MIKE.MFBSpace  
DROP DBSPACE PUBLIC.SPAC10
```

Authorization:

A DBSPACE may be dropped only by its owner or by a user having DBA authority. You must have DBA authority to drop a PUBLIC DBSPACE. No user, even with DBA authority, can drop the DBSPACE containing the SQL/DS catalogs.

The DROP DBSPACE statement destroys the contents of a DBSPACE and returns the DBSPACE to an "available" state. The DROP DBSPACE statement is a much faster way to destroy the contents of a DBSPACE than by deleting the data one row at a time or one table at a time. You can use DROP DBSPACE with both PUBLIC and PRIVATE DBSPACES.

All existing access modules for programs that operate on the dropped DBSPACE are automatically marked "invalid." If one of these programs is currently running,

SQL/DS does not drop the DBSPACE until the running program ends its current logical unit of work. The invalid access modules remain in the data base until they are explicitly dropped via a DROP PROGRAM statement (described in the previous chapter). When an invalid access module is invoked, SQL/DS attempts to regenerate it and restore its validity. However, if the program contains any SQL statement that refers to a DBSPACE or table that has been dropped, that SQL statement returns a negative SQLCODE at execution time.

Changing DBSPACE Characteristics

Format:

```
ALTER DBSPACE [owner.]dbspace-name ( { PCTFREE = integer }  
                                     { LOCK = size } )
```

The LOCK parameter is applicable to PUBLIC DBSPACEs only.

Examples:

```
ALTER DBSPACE MFBSpace (PCTFREE = 0)  
ALTER DBSPACE PUBLIC.SPACe (LOCK = PAGE,PCTFREE=3)
```

Authorization:

To alter a PRIVATE DBSPACE, you must either own the DBSPACE, or have DBA authority. You must have DBA authority to alter a PUBLIC DBSPACE.

The ALTER DBSPACE statement allows you to alter the percentage of free space that SQL/DS reserves on each data page when records are inserted into a PUBLIC or PRIVATE DBSPACE. It also allows you to alter the lock size of a PUBLIC DBSPACE. (You can't alter the lock size of a PRIVATE DBSPACE.)

When you acquire a DBSPACE, you should set the percentage of free space to some number greater than zero (the default is 15 percent). A typical use of ALTER DBSPACE is to set the percentage of free space to zero (PCTFREE=0) after initial loading of data into a DBSPACE. Subsequent insertions can then take advantage of the free space that SQL/DS reserves during the loading process. It is also possible to increase PCTFREE again for a later loading phase.

To alter the lock size of a PUBLIC DBSPACE at any time, use the LOCK parameter. You can specify both the PCTFREE and LOCK parameters when altering a PUBLIC DBSPACE. If you specify both, you can specify them in any order, but you must separate them with a comma. (See the example above.) The valid lock sizes are ROW, PAGE, and DBSPACE, as described under the ACQUIRE DBSPACE statement. When an ALTER DBSPACE statement is

executed to alter the lock size of a DBSPACE, SQL/DS acquires an exclusive lock on the entire DBSPACE and holds the lock until the end of the current logical unit of work. The newly selected lock size then becomes effective for subsequent logical units of work.

SQL/DS Automatic Locking Mechanism

Recall from the *SQL/Data System Concepts and Facilities for VM/SP* manual that SQL/DS can operate in either multiple user or single user mode. When you use SQL/DS in single user mode, there is no contention from other users when you attempt to access data. In multiple user mode, however, other users may be accessing data that you are trying to access. SQL/DS provides for concurrent access via a locking mechanism.

Internally, SQL/DS acquires locks on data accessed by a logical unit of work. There are two types of locks, called *share locks* and *exclusive locks*. All logical units of work automatically acquire exclusive locks on all data that they modify and acquire share locks on data that they read. Exclusive locks prevent other users from reading or modifying the data. Share locks permit other users to read, but prevent them from modifying, the data. In general, locks are held to the end of the logical unit of work in which they are acquired.

SQL/DS automatically detects and corrects potential deadlocks. A deadlock occurs when two logical units of work are each waiting to access data that the other has locked. Fortunately, SQL/DS detects these situations and “backs out” the youngest logical unit of work. A “back out” means that SQL/DS restores all changes made to the data base during that logical unit of work, and then releases the lock that was acquired for it. The other application can then proceed. If your logical unit of work is backed out, SQL/DS returns a negative SQLCODE and sets on a warning flag (SQLWARN6).

SQL/DS locking is fully automatic and requires no user intervention. However, certain statements permit knowledgeable users to adjust or override the normal locking. The size of the lockable data units can be adjusted by the LOCK option of the ACQUIRE DBSPACE and ALTER DBSPACE statements. Also, you can override automatic locking and explicitly acquire certain kinds of locks by the LOCK statement. All of these statements are discussed in the following sections.

The only way to run SQL/DS with locking inhibited is in single user mode.

Ending the Logical Unit of Work

When you end a logical unit of work, you must tell SQL/DS what to do with changes made to the data. The data changes can either be saved (“committed”) or ignored (“rolled back”). The COMMIT WORK and ROLLBACK WORK commands in your program tell SQL/DS to either save the data changes or ignore them.

COMMIT WORK

Format:

```
COMMIT WORK [RELEASE]
```

Authorization:

Anyone connected to SQL/DS can issue this statement.

The COMMIT WORK statement ends the current logical unit of work if one is in progress. SQL/DS commits any changes made during the logical unit of work to the data base.

It is strongly recommended that each application program explicitly end its logical unit of work before terminating. If you don't explicitly end the logical unit of work, SQL/DS automatically commits (upon successful termination of the program) all changes made by the program during its pending logical unit of work.

Note: If you don't specify COMMIT WORK in SQL/DS Release 1 single user mode batch applications, SQL/DS rolls back the changes. This exception is removed in SQL/DS Release 2.

See "Application Epilog" on page 93 and "Error Handling" on page 202 for more information about program termination.

If you use the RELEASE option in a VM/SP environment, your default userid established by implicit connect is re-established for a subsequent logical unit of work. If you had overridden this default userid with an explicit CONNECT in the terminating logical unit of work, that explicitly established userid is replaced by the default userid. If the RELEASE option is omitted, the userid in effect at termination of the logical unit of work is retained for a subsequent logical unit of work. (See "VM/SP Connect Considerations" on page 186.)

COMMIT WORK has no effect on the contents of host variables or on the control flow of the host program.

ROLLBACK WORK

Format:

ROLLBACK WORK [RELEASE]

Authorization:

Anyone connected to SQL/DS can issue this statement.

The ROLLBACK WORK statement restores the data base to its state prior to the current logical unit of work. SQL/DS terminates the current logical unit of work (if any).

If you use the RELEASE option in a VM/SP environment, your default userid established by implicit connect is re-established for a subsequent logical unit of work. If you had overridden this default userid with an explicit CONNECT in the terminating logical unit of work, that explicitly established userid is replaced by the default userid. If the RELEASE option is omitted, the userid in effect at termination of the logical unit of work is retained for a subsequent logical unit of work. (See "VM/SP Connect Considerations" on page 186.)

ROLLBACK WORK has no effect on the contents of host variables or on the control flow of the host program.

Under some circumstances, SQL/DS automatically backs out of a logical unit of work. See "SQL/DS Automatic Locking Mechanism" on page 232 for more information.

Note: If you use a ROLLBACK WORK in a routine that was entered because of an error or warning and you used the SQL WHENEVER statement, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK. This avoids a program loop if the ROLLBACK WORK fails with an error or warning.

Using the LOCK Statement to Override Automatic Locking

Format:

```
LOCK {TABLE [creator.]table-name|DBSPACE [owner.]dbspace-name}  
    IN {SHARE|EXCLUSIVE} MODE
```

Examples:

```
LOCK TABLE PARTS IN EXCLUSIVE MODE  
LOCK DBSPACE DSP3 IN SHARE MODE
```

Authorization:

To lock a DBSPACE, you must either be the owner of the DBSPACE, or have DBA authority. You can lock a table if you own the table, if you have the SELECT privilege on the table, or if you have DBA authority. No user, regardless of authority, can lock any SQL/DS catalog or the DBSPACE containing the catalogs.

The LOCK statement overrides the SQL/DS automatic locking mechanism. It explicitly acquires a lock on a table or DBSPACE. SQL/DS holds the requested lock until the end of the current logical unit of work.

Naturally, the LOCK command is useful only in multiple user mode. In single user mode, there is no contention for resources, and, hence, no locking. When running in single user mode, SQL/DS ignores all LOCK statements.

An *exclusive* lock prevents other users from reading or changing any data in the locked table or DBSPACE. A *share* lock permits other users to read, but prevents them from modifying, the data in the locked object.

The requested lock may be unavailable because other logical units of work are reading or modifying the indicated data. When this is the case, the logical unit of work that requested the lock waits until the other active logical units of work have completed. SQL/DS then grants the lock and the requesting logical unit of work proceeds normally.

Note that it is *never* necessary for you to use the LOCK statement; SQL/DS has fully automatic locking. You may issue all SQL queries and updates independently of explicit LOCK statements.

The LOCK statement is useful mainly for avoiding the overhead of acquiring many small locks when scanning over a table. For example, suppose some DBSPACE has been acquired with a lock size of row. If you know that you will be accessing all the rows of a table within that DBSPACE, you may explicitly lock the table to avoid the overhead of acquiring a lock on each individual row.

A LOCK statement on a table in a PRIVATE DBSPACE is the same as a LOCK statement on the entire DBSPACE, since locking is always done at the DBSPACE level for PRIVATE DBSPACES.

Use of Long Fields

Use of the LONG VARCHAR and LONG VARGRAPHIC data types is subject to more severe limitations than other data types in SQL/DS. Long fields are intended for storage of unstructured data such as text strings, images, and drawings. Data that you intend to use in search conditions should not be placed in such fields.

You can use either a CREATE TABLE or ALTER TABLE statement to create a long field. As with other data types, you can disallow null values by specifying the NOT NULL option when creating the field. See the CREATE TABLE and ALTER TABLE statements for more information about defining tables and fields.

These are the only operations that SQL/DS permits on long fields:

1. SELECT in an outer-level query (not in a subquery).
2. INSERT into the data base from an input host variable (not from a constant or from a subquery). You can, however, insert null values into long fields via the usual INSERT statement mechanisms. (That is, you are not restricted to host variables when inserting nulls.)
3. UPDATE from an input host variable or UPDATE to the null value. (SET LONGFIELD=:X and SET LONGFIELD=NULL are permitted, but SET LONGFIELD='HELLO' and SET LONGFIELD=OTHERFIELD are not permitted.)
4. DELETE of rows containing long fields.

You must observe these rules when using long fields:

1. No indexes are permitted on long fields.
2. A long field cannot be an operand of ORDER BY or GROUP BY or DISTINCT or a built-in function. A query involving ORDER BY can select a long field only if the long field is not included in the ORDER BY clause.
3. Predicates (search conditions) are not permitted on long fields.
4. Queries that are part of a UNION cannot select long fields.
5. Long fields may be used in views if their usage does not conflict with the above rules. (Views are discussed next.)

Updating Internal Statistics

Format:

```
UPDATE [ALL] STATISTICS FOR
  {TABLE [creator.]table-name | DBSPACE [creator.]dbspace-name}
```

Example:

```
UPDATE STATISTICS FOR TABLE QUOTATIONS
```

Authorization:

No authorization is required for this statement. Any user can issue UPDATE STATISTICS for any table or DBSPACE.

The UPDATE STATISTICS statement is used to bring up to date the internal statistics recorded by SQL/DS for a table and its indexes. These statistics, which are contained in the SQL/DS catalogs, include the size of the table, various index characteristics, and other information. The SQL/DS preprocessor uses these statistics when choosing access paths for SQL statements. If UPDATE STATISTICS is not issued for a table, the SQL/DS preprocessor uses default values in choosing access paths to the table. These default values may not be as efficient as those generated by UPDATE STATISTICS.

You should invoke the UPDATE STATISTICS statement for a table after a significant number of changes have been made to the data in the table since it was last examined by UPDATE STATISTICS. For example, you might want to issue UPDATE STATISTICS after a table has been changed by 20 percent or more.

The UPDATE STATISTICS statement scans over the indicated table and each of its indexes; therefore, it is a relatively expensive and time-consuming statement. If the ALL option is specified, statistics are updated for all columns including those that contain indexes. In the case of columns without indexes, the column statistics are an approximation. If ALL is not specified, statistics are updated for only the columns that contain indexes.

If the DBSPACE option is selected, statistics are updated for every table in the designated DBSPACE.

Data Definition

SQL Data Definition statements allow you to:

- Create and drop tables (CREATE TABLE and DROP TABLE)
- Create and drop indexes on tables (CREATE INDEX and DROP INDEX)

- Add new columns to existing tables (ALTER TABLE)
- Create and drop synonyms for table names (CREATE SYNONYM and DROP SYNONYM)
- Enter comments about tables into the SQL/DS catalogs (COMMENT).

One advantage of SQL/DS is that you may define new objects in the data base without stopping the system or invoking special utilities. Thus, for example, your application program can create a table for storing and manipulating some temporary result, and drop the table when it is no longer needed.

Data Definition statements automatically update the SQL/DS catalogs that describe the data base. (These catalogs are explained more completely in the *SQL/Data System Planning and Administration for VM/SP* manual.) To avoid inconsistently updating the catalogs, SQL/DS rolls back the current logical unit of work if it encounters certain types of errors related to Data Definition statements. An error code lets you know if a logical unit of work is rolled back. **You should put Data Definition statements in a logical unit of work of their own to minimize the effects of an automatic roll back.**

Some Data Definition statements may invalidate the access modules of one or more programs previously preprocessed by SQL/DS. If the access module for a program named PLANNER, for example, accesses the INVENTORY table by a particular index, SQL/DS renders this access module invalid if someone drops that index. In this case, when PLANNER is next used, SQL/DS creates a new access module based on the indexes currently available. No changes need be made to PLANNER. The process of producing the new access module is entirely transparent to programs except for a slight delay in processing the first SQL statement.

Creating a Table

Format:

```
CREATE TABLE [creator.]table-name
  (column-name-1 data-type-1 [NOT NULL]
  [,column-name-2 data-type-2 [NOT NULL] ] ... )
  [IN [owner.]dbspace-name]
```

Example:

```
CREATE TABLE MIKE.MUSICIANS
  ( NAME          VARCHAR(20) NOT NULL,
    INSTRUMENT    VARCHAR(10) NOT NULL,
    "BAND NAME"   VARCHAR(10)
  )
  IN MIKE.DBSP1
```

Authorization:

You must have RESOURCE authority to create a table, unless someone with DBA authority has acquired a PRIVATE DBSPACE on your behalf. If a DBA does acquire a DBSPACE for you, you may create a table in that DBSPACE, even if you do not have RESOURCE authority. You must have DBA authority to create a table for another user.

This statement creates a new table in the data base; the table has the designated name and the designated columns. The table-name must follow the rules for an SQL identifier as discussed under “General Rules for Naming Data Objects” on page 74.

If you specify the NOT NULL option for a column of a table, SQL/DS does not permit null values in that column. Any statement that attempts to place a null value in such a column is rejected with an error code.

Before declaring a column DECIMAL with a scale of 0, you should consider declaring it INTEGER or SMALLINT instead. SMALLINT and INTEGER data types use storage more efficiently than a DECIMAL value with a scale of 0.

Once a table has been created, you may not change the data types of its columns or drop a column from the table. However, you may add new columns to the table by the ALTER TABLE statement.

Each table in the data base has a *creator*, which is noted in the SQL/DS catalogs. If creator isn't specified, the creator of a table is the user who preprocessed the program in which the table is created. (Certain exceptions to this rule are explained under “Dynamically Defined Statements” on page 147.)

For example, if user SCOTT preprocesses a program that creates a table named SUMMARY, and user JONES runs the program, the creator of the SUMMARY table is SCOTT. Note that JONES must be a DBA to run the program. Any program preprocessed by SCOTT can refer to the SUMMARY table simply by the name SUMMARY. Any program preprocessed by another user that refers to the SUMMARY table must use Scott's userid as a prefix to the table-name, SCOTT.SUMMARY. (Before the user can run the program successfully, though, Scott must grant the user the proper privileges.) A table-name qualified by its creator can be used in any kind of SQL statement. A given user cannot create two tables with the same name; however, two different creators may have tables with the same name.

A newly created table is placed in one of the existing DBSPACES of the data base according to the following rules:

1. If you specify a dspace-name in the CREATE TABLE statement, the table goes into the named DBSPACE. The owner of the DBSPACE must be either the user who preprocessed the current program, or PUBLIC. If you have DBA authority, you can create a table in a PRIVATE DBSPACE of any user by qualifying the dspace-name with its owner's userid, as follows:

```
CREATE TABLE .... IN SCOTT.DSP3
```

2. If you don't specify a dbspace-name in the CREATE TABLE statement, the table goes into any PRIVATE DBSPACE owned by the user who preprocessed the program. In the case where:
 - a. The person who preprocessed the program has DBA authority, and
 - b. No DBSPACE is specified, and
 - c. The table name is qualified with a userid,

SQL/DS places the table into any PRIVATE DBSPACE owned by the specified userid. If there is no such DBSPACE, an error condition results.
3. If the dbspace-name is not qualified, SQL/DS will not place the table into a nonrecoverable DBSPACE by default. Therefore, if you wish to create a table in a nonrecoverable DBSPACE, you must specify the dbspace-name.
4. If both the table name and the DBSPACE name are qualified, but are not qualified with the same userid, and the user who preprocessed the program has DBA authority, SQL/DS uses both qualifiers. That is, if JIM has DBA authority, he may create table KELLI.SUPPLIERS in DBSPACE JOE.SPACE1.

Figure 26 summarizes where a table is placed depending on what is specified. X represents the userid of the person who preprocessed the program. X is denoted as optional below because if no userid is specified, the creator always defaults to the userid of the person who preprocessed the program (X). Y represents some other userid.

User X Preprocesses the Program:					
DBA Needed?	Table Creator	Table Name	DBSPACE Owner	DBSPACE Name	SQL/DS Action
NO	[X]	A			User X creates X.A in a PRIVATE DBSPACE owned by X.
YES	Y	A			User X creates Y.A in any PRIVATE DBSPACE owned by Y.
NO	[X]	A	[X]	B	User X creates X.A in X.B ¹
YES		A	Y	B	User X creates X.A in Y.B
YES	Y	A		B	User X creates Y.A in Y.B ¹
YES	Y	A	Z	B	User X creates Y.A in Z.B

¹ If there is no PRIVATE DBSPACE, B, but there is PUBLIC DBSPACE, B, the PUBLIC DBSPACE will be used.

Figure 26. Default Table Placement

Confusion can be easily avoided if you always concatenate the desired userids to both the table name and the DBSPACE name. When you do this, there is no doubt who you want to create the table for and where you want the table to be placed; the only concern is whether you are authorized to create a table for someone else, or in someone else's DBSPACE.

Adding a Column to a Table

Format:

```
ALTER TABLE [creator.]table-name  
    ADD column-name data-type
```

Example:

```
ALTER TABLE SCOTT.SUPPLIERS ADD RATING CHAR(1)
```

Authorization:

To alter a table, you must either be its creator or have the ALTER privilege on it. You can alter any table if you have DBA authority (even the SQL/DS catalogs).

This statement expands an existing table by adding one new column on the right-hand side of the table. All existing rows of the table are expanded and considered to have a null value for the new column. Therefore, you cannot specify the NOT NULL option for the new column. Also, the new column-name must not be the same as the name of any existing column in the table. The data-type can be any of the SQL/DS data types listed earlier.

Once a column name is established, it remains for the life of the table. To change it, you must drop the table and re-create it.

Dropping a Table

Format:

```
DROP TABLE [creator.]table-name
```

Example:

```
DROP TABLE INVENTORY
```

Authorization:

You can drop a table only if you have created the table, or if you have DBA authority.

This statement drops the indicated table from the data base. All indexes and views defined on the table, and all privileges granted on the table, are also dropped from the data base. All contents of the table are lost. However, users can have previously defined synonyms (via a CREATE SYNONYM statement) for the name of the table that was dropped; these synonyms remain in effect even though the data no longer exists.

When a table is dropped, SQL/DS marks invalid any access modules for programs that operate on the dropped table. The invalid access modules remain in the data base until they are explicitly dropped by a DROP PROGRAM statement. When an SQL statement attempts to invoke an invalid access module, SQL/DS tries to regenerate the module to restore its validity. However, if the SQL statement refers to a dropped DBSPACE or table, that SQL statement returns an error code at execution time.

If you issue a DROP TABLE statement while some program that depends on the table is running and has a logical unit of work in progress, the DROP TABLE statement does not take effect until the end of the running logical unit of work. Meanwhile, your program waits.

When dropping a table, SQL/DS temporarily requires additional space so it can restore the table in case the logical unit of work is not committed. SQL/DS behaves as though a table approximately doubles in size immediately before it is dropped. The empty pages are taken from the DBSPACE from which the table was dropped. Note that if all rows of a table have previously been deleted, such additional space is not required.

Creating an Index

Format:

```
CREATE [UNIQUE] INDEX [creator.]index-name ON [creator.]table-name
  ( column-name-1 [ ASC | DESC ]
    [,column-name-2 [ ASC | DESC ] ] ... )
  [PCTFREE= {10|integer} ]
```

Example:

```
CREATE INDEX FASTQUOTES ON QUOTATIONS
(PARTNO ASC, PRICE DESC, DELIVERY_TIME)
PCTFREE = 33
```

Authorization:

You can create indexes on tables that you have created. You need the INDEX privilege or DBA authority to create an index on another user's table (for example, CREATE INDEX MYID.IND1 ON OTHERID.TAB1). You need DBA authority to create an index *for* another user (for example, CREATE INDEX OTHERID.IND1 ON OTHERID.TAB1).

This statement allows you to create an index on one or more columns of a table, and give a name to the new index. The indicated table must exist, but it may be empty. None of the columns over which the index is created may be of type LONG VARCHAR or LONG VARGRAPHIC.

You can create an index on a column in either ascending (ASC) or descending (DESC) order. Ascending order is the default. Performance may be improved for queries that access the indexed column in the specified order.

An index is maintained by SQL/DS until it is explicitly dropped in a DROP INDEX statement, or until its table or DBSPACE is dropped.

Indexes are invisible to application programs in the sense that SQL/DS provides no means for using an index directly. The SQL/DS preprocessor chooses which index, if any, is to be used in processing a given query or data manipulation statement. However, the existence of an index has the following implications:

1. An index on a column of a table, such as the PARTNO column of the QUOTATIONS table, provides SQL/DS with a fast means to access the table directly by the indexed column. This improves the performance of queries based on that column, such as searching for rows of QUOTATIONS with a given PARTNO. However, there is a slight increase in the time required to update the indexed column, since SQL/DS must update the index also.

If you are going to do many updates to an indexed column, or are going to insert many rows into an indexed table (as the Data Base Services utility does), consider dropping the index before you do the updates and then re-creating it after the updates are complete. This allows SQL/DS to update the table without having to update the index.

2. If you declare an index UNIQUE when you create it, SQL/DS ensures that no two rows of the indicated table are identical in the indexed column(s). For example, you can ensure that no two rows of the SUPPLIERS table have the same SUPPNO by creating a unique index on SUPPLIERS(SUPPNO). You can force all rows of a table to be unique by creating a unique index on all columns of the table. Any INSERT or UPDATE statement that would cause a table to violate the uniqueness property of an index fails with an error code. If,

when a CREATE UNIQUE INDEX statement is executed, the table already contains some rows that are not unique in the indexed columns, the CREATE UNIQUE INDEX statement fails and returns an error code.

As in the CREATE TABLE statement, the user who preprocessed the program that creates an index becomes the *creator* of the index. (Certain exceptions to this rule are explained under “Dynamically Defined Statements” on page 147.) A given creator cannot have two indexes with the same name, but two different creators may each have an index with the same name. Only the creator of an index (or a user with DBA authority) can drop the index.

The optional PCTFREE clause of a CREATE INDEX statement controls the amount of free space reserved in an index for later insertions and updates. PCTFREE defines the percentage of the total space of the index that is to be reserved for this purpose. It may range from 0 to 99, but for practical purposes should not exceed 50. Increasing PCTFREE causes the index to take more space in the data base, but reduces the time required to insert or update rows of the indexed table. If you don't include a PCTFREE clause in the CREATE INDEX statement, SQL/DS sets PCTFREE to the default value of 10.

When creating indexes on multiple columns, you must observe the following limitations:

1. An index cannot be created on more than 16 columns.
2. The sum of the widths of the indexed columns, plus 25% of the widths of any indexed columns of varying-length character type, must not exceed 255 bytes. If you are creating the index after data has been loaded into the table, an SQL sort is invoked during the preprocessing of the CREATE INDEX command. If duplicate keys are allowed in the index, then the sort will require 4 bytes to be added to the encoded key. These four bytes are part of the 255 total bytes.

Also, when creating new indexes, remember the following rules:

1. When you preprocess a program, SQL/DS creates an access module for it that takes advantage of the best access path available at the time of the preprocessing. Therefore, it is good practice to create indexes *before preprocessing* programs that might take advantage of them.
2. When you create a new index, existing access modules are not made invalid because they can still use their original access path. However, an existing program may run more efficiently by taking advantage of the new index. If this is the case, you should preprocess the program again. A new access module is then created for the program, possibly using the new index.

Dropping an Index

<p><i>Format:</i></p> <pre>DROP INDEX [creator.]index-name</pre>
<p><i>Example:</i></p> <pre>DROP INDEX FASTQUOTES</pre>
<p><i>Authorization:</i></p> <p>You can use the DROP INDEX statement only if you are the creator of the index to be dropped (or if you have DBA authority).</p>

The DROP INDEX statement drops the indicated index from the data base. The table on which the index is defined is not affected.

If you have DBA authority, you can drop another user's index by qualifying the index name with that person's userid (for example, SMITH.TABINDEX).

All existing access modules that use the dropped index are marked invalid. A subsequent attempt to use one of these access modules causes SQL/DS to re-create the access module, using the best access paths currently available. The process of producing the new access module is entirely transparent to your program except for a slight delay in processing your first SQL statement.

Creating a Synonym

<p><i>Format:</i></p> <pre>CREATE SYNONYM identifier FOR creator. {table-name view-name}</pre>
<p><i>Example:</i></p> <pre>CREATE SYNONYM PARTS FOR SMITH.INVENTORY</pre>
<p><i>Authorization:</i></p> <p>Anyone connected to SQL/DS can issue this statement. You can create a synonym for any user's table or view. (No authorization is required.)</p>

The CREATE SYNONYM statement defines an alternative name for a table or view. For example, the statement:

```
CREATE SYNONYM PARTS FOR SMITH.INVENTORY
```


defines the alternative name PARTS to refer to the table named INVENTORY whose owner is SMITH. The right-hand side of the CREATE SYNONYM statement (SMITH.INVENTORY in the above example) must be the name of a table or view, not another synonym.

Synonyms are commonly used when a group of users all wish to share a table. Suppose one user, ADAMS, creates a table called DATA. All users wishing to share this table can then issue the statement:

```
CREATE SYNONYM DATA FOR ADAMS.DATA
```

Then each user can refer to the shared table as DATA, without using the fully qualified name ADAMS.DATA. (Remember that ADAMS must authorize the other users to access his table.)

A synonym is effective only for the user who created it. If many users wish to have the same synonym, they must each issue a CREATE SYNONYM statement.

When synonyms are created in a program, the creator is the user who preprocessed the program. (Certain exceptions are described under "Dynamically Defined Statements" on page 147.) More commonly, synonyms are created via ISQL or the DBS utility.

Once created, a synonym remains in effect until it is explicitly dropped by a DROP SYNONYM statement.

Dropping a Synonym

Format:

```
DROP SYNONYM identifier
```

Example:

```
DROP SYNONYM DATA
```

Authorization:

You can drop a synonym only if you have created it.

The DROP SYNONYM statement causes the indicated synonym to be dropped from the data base. The table on which the synonym is based is not affected.

Dropping a synonym does not affect the access modules of existing programs that use the synonym, since in the access modules the synonym has already been resolved to a real table name. However, a program containing a dropped synonym cannot be preprocessed successfully, either automatically or by user request.

Putting Comments into SQL/DS Catalogs

Format 1:

```
COMMENT ON { TABLE [creator.]table-name           } IS quoted-string
           { COLUMN [creator.]table-name.column-name }           }
```

Format 2:

```
COMMENT ON [creator.]table-name (column-name IS quoted-string,
.....;
column-name IS quoted-string)
```

Examples:

```
COMMENT ON COLUMN QUOTATIONS.DELIVERY_TIME IS 'MEASURED IN DAYS'
```

```
COMMENT ON TABLE INVENTORY IS 'INVENTORY OF MACHINE PARTS ONLY'
```

```
COMMENT ON QUOTATIONS (SUPPNO IS 'SEE SUPPLIERS TABLE FOR NAMES',
PRICE IS 'IN U.S. DOLLARS', DELIVERY_TIME IS 'MEASURED IN DAYS')
```

Authorization:

You may use the **COMMENT** statement only if you are the creator of the table in question or if you have DBA authority.

The SQL **COMMENT** statement lets you associate remarks or comments with your tables or views, or with columns in your tables or views. SQL/DS places the comment that you specify into one of the SQL/DS catalogs. The comment is put into either:

1. The **REMARKS** column of **SYSCATALOG** (if you are commenting on a table or view); or,
2. The **REMARKS** column of **SYSCOLUMNS** (if you are commenting on a column).

If there already is a comment for your table or column, the new comment replaces the old one.

The basic format of the **COMMENT** statement is as follows:

```
COMMENT ON TABLE [creator.]table-name IS quoted-string
or
COMMENT ON COLUMN [creator.]table-name.column-name IS quoted-string
```

For instance, the statement:

```
COMMENT ON COLUMN QUOTATIONS.DELIVERY_TIME IS 'MEASURED IN DAYS'
```

finds the row in SYSCOLUMNS which is associated with the DELIVERY__TIME column of the QUOTATIONS table. SQL/DS then inserts the explanatory comment 'MEASURED IN DAYS' into the REMARKS of that row. The next time you did a SELECT on SYSCOLUMNS, you would see the 'MEASURED IN DAYS' in the REMARKS column of the DELIVERY__TIME row.

For another example, if you were to enter the statement:

```
COMMENT ON TABLE INVENTORY IS 'INVENTORY OF MACHINE PARTS ONLY'
```

SQL/DS would place the comment 'INVENTORY OF MACHINE PARTS ONLY' into the REMARKS column of the SYSCATALOG table. This would be on the row associated with your INVENTORY table.

Another format for the COMMENT statement is available. This format can be used to specify comments for more than one column in the same table in one statement. It can be used only for columns, not for tables or views. The format for this statement is:

```
COMMENT ON [creator.]table-name (column-name IS quoted-string,  
.....,  
column-name IS quoted-string)
```

For instance, if you entered the statement:

```
COMMENT ON QUOTATIONS (SUPPNO IS 'SEE SUPPLIERS TABLE FOR NAMES',  
PRICE IS 'IN U.S. DOLLARS', DELIVERY__TIME IS 'MEASURED IN DAYS')
```

SQL/DS would place all three comments into the REMARKS column of the SYSCOLUMNS catalog. The next time you did a SELECT on that catalog, you would see 'SEE SUPPLIERS TABLE FOR NAMES' on the row associated with the SUPPNO column of the QUOTATIONS table. You would also see 'IN U.S. DOLLARS' on the PRICE row, and 'MEASURED IN DAYS' on the DELIVERY__TIME row.

To use the COMMENT statement, you must be the creator of the table in question, or you must have DBA authority. You must have SELECT authority on the SYSTEM.SYSCATALOG and SYSTEM.SYSCOLUMNS catalog tables to see the comments that you put in them.

The comment must not exceed 254 characters, and must be enclosed in single quotes, as is shown above. To represent a single quote within a comment, use two single-quotes:

```
COMMENT ON TABLE SUPPLIERS IS  
    'DON''T ADD NEW SUPPLIERS UNTIL THE CONTRACTS  
    ARE APPROVED'
```

You can comment on another user's table only if you have DBA authority. To comment on another user's table, concatenate the userid to the table name in the usual fashion:

```
COMMENT ON COLUMN JONES.EMPTABLE.NAME IS ....
```

Putting Labels on Tables or Columns

Format 1:

```
LABEL ON { TABLE [creator.]table-name           } IS quoted-string  
         { COLUMN [creator.]table-name.column-name }           }
```

Format 2:

```
LABEL ON [creator.]table-name (column-name IS quoted-string,  
                               .....  
                               column-name IS quoted-string)
```

Examples:

```
LABEL ON TABLE QUOTATIONS IS 'CURRENT PRICE QUOTATIONS'
```

```
LABEL ON COLUMN INVENTORY.PARTNO IS 'PART NUMBER'
```

```
LABEL ON INVENTORY (PARTNO IS 'PART NUMBER', DESCRIPTION IS  
'PART DESCRIPTION', QONHAND IS 'QUANTITY ON HAND')
```

Authorization:

Only the creator of the table or a user with DBA authority can issue the LABEL statement on a table or column.

The SQL LABEL ON statement lets you define a label for a table name or a column name. Unlike synonyms, labels cannot be used as identifiers. Instead, they can be used in displays created by applications that process SQL commands dynamically. You can enter SQL commands using the actual table and column names (which are easier to enter). The program can display the results using the labels (which are easier to understand) instead of the table and column names.

Labels are ignored by DBS utility and ISQL SELECT processing. Only column names will identify SQL SELECT command output displayed by DBS utility or ISQL processing.

The basic format of the LABEL ON statement is as follows:

```
LABEL ON TABLE [creator.]table-name IS quoted-string
```

or

```
LABEL ON COLUMN [creator.]table-name.column-name IS quoted-string
```

For instance, the statement:

```
LABEL ON TABLE SUPPLIERS IS 'SUPPLIER NAMES AND ADDRESSES'
```

defines a label of 'SUPPLIER NAMES AND ADDRESSES' for the SUPPLIERS table. This could then be used as a universal heading in all the reports created by various applications using this table. The statement:

```
LABEL ON COLUMN INVENTORY.QONHAND IS 'QUANTITY ON HAND'
```

defines a common presentation heading of 'QUANTITY ON HAND' for the QONHAND column of the INVENTORY table.

An additional format for the LABEL ON statement is available to let you create labels on more than one column in a table at the same time. Similar to the second format of the COMMENT command (discussed above), this format is as follows:

```
LABEL ON [creator.]table-name (column-name IS quoted-string,  
.....,  
column-name IS quoted-string)
```

If you entered the statement:

```
LABEL ON INVENTORY (PARTNO IS 'PART NUMBER', QONHAND IS  
'QUANTITY ON HAND')
```

you would define labels for both the PARTNO and QONHAND columns of the INVENTORY table. This format cannot create a label for a table.

Labels for tables are stored in the TLABEL column of the SYSTEM.SYSCATALOG catalog table; labels for columns are stored in the CLABEL column of the SYSTEM.SYSCOLUMNS catalog table. You can see what labels are defined by querying these catalogs.

Column labels are returned in the SQLDA when a "SELECT" statement is described by a "DESCRIBE" statement. The USING clause of the DESCRIBE statement tells SQL/DS whether you want to have the column names or the column labels (or both) returned. See "DESCRIBE" on page 286 for more information on returning labels with the DESCRIBE statement. Your program can then move the label from the SQLNAME field of the SQLDA into a work area.

A column is considered to have no label if either its LABEL column in SYSTEM.SYSCOLUMNS is NULL, or if it has a zero length value. If there is no column label when you try to do a DESCRIBE, the SQLNAME field of the SQLDA is set to length 0, and the field is cleared to 30 blanks. For this reason, your program should move the label into a work area using the length returned in SQLDA only after it makes sure that the length is not zero.

Performance Considerations

When preprocessing your program, there are two performance parameters that you can specify: the BLOCK/ NOBLOCK option and the isolation level option. The format and use of these options within the SQLPREP EXEC was discussed under "Preprocessing the Program" on page 187. This section is devoted to showing you when you would want to specify each of these options.

Selecting the Isolation Level

SQL/DS puts locks on data that your program is working with, to keep other users from reading or changing that data. You can specify how long SQL/DS will hold the lock on data that your program has already read. You can tell SQL/DS either to lock **all** the data that the current logical unit of work has read, or to lock just the row of data that a cursor is currently pointing to. This is called specifying the “isolation level” of the lock. If you choose to put a lock on all the data that your program’s current logical unit of work has read, this is called specifying *isolation level repeatable read*. Repeatable read locks are held until the end of the logical unit of work. If you choose to just put a lock on the row or page of data that your cursor is pointing to, then you are specifying *isolation level cursor stability*. With cursor stability locking, when the cursor moves, SQL/DS frees all the data previously read by the program and held by the lock.

Both repeatable read and cursor stability provide you with the following data isolation from other concurrent users:

1. Your LUW (logical unit of work) cannot modify or read any data which another active LUW has modified. Similarly, if your LUW has modified some data, no one else can modify or read that data until your LUW has ended. Modify implies SQL INSERT, DELETE, UPDATE, or PUT. Read implies SQL SELECT or FETCH.
2. If your LUW has a cursor pointing to a row of data, no other LUW can modify that data. Similarly, your LUW cannot modify a row which another user has a cursor pointing to.

In addition to the above, repeatable read locking provides you with the following data isolation from other concurrent users:

1. No other LUW can modify any row which your active LUW has read. Also, you cannot modify any data which another active LUW, specifying repeatable read, has read.
2. You don’t have to worry about your data being changed between reads, as long as you don’t end your LUW between those reads.

This extra isolation has its drawbacks. When you specify repeatable read for data in PUBLIC DBSPACES with PAGE or ROW level locking, you reduce the concurrency of the data. This means that other users may be locked out from the data for a long time, causing delays in their programs’ executions.

Isolation level cursor stability reduces these locking problems by making the data more available. When you specify cursor stability, SQL/DS does not hold the locks as long. Once a cursor has moved past a row or page of data, the lock on that data is dropped. This increases concurrency. Other users can access data faster while, at the same time, no other user can modify the row or page of data pointed to by your cursor.

Cursor stability can, however, cause some data inconsistencies. For instance, if an LUW reads data twice, it can get different results each time. Another user could have modified the data and committed the modification. Also, your program could

try to re-read a row of data and find it doesn't exist any longer. Another user could have deleted the row during your logical unit of work, and committed the change.

When should each of these options be chosen? Usually, you should specify repeatable read locking for your programs. You should only use cursor stability if your program causes or will cause locking problems. For instance, you would probably want to use cursor stability for transactions that perform terminal reads without performing a COMMIT or ROLLBACK WORK. You probably would also want to use cursor stability for programs that do bulk reading. It is handy for programs that browse through large amounts of data. On the other hand, programs that perform commits or rollbacks before issuing terminal reads should use repeatable read locking, since they probably will not cause locking problems. Also, some applications need to protect themselves against updates. These programs should also use repeatable read locking.

You can also "mix" isolation levels. Your program can set, change and control its own isolation level as it is running. You can specify mixed isolation level with the USER option of the ISOLATION preprocessor parameter, as detailed under "Preprocessing the Program" on page 187.

If you choose this option, your program must pass the isolation level value to SQL/DS via a program variable. It **must** declare a one-character program variable and **must** set this variable to the desired isolation level value before executing SQL statements. To set the isolation level to repeatable read, your program should set this variable to 'R'. For cursor stability, the variable should be set to 'C'. The program can change the variable at any time so that the subsequent SQL statements will be executed at the new isolation level value. However, if your program changes the isolation level while a cursor is OPEN, all operations on that cursor (until the cursor has been closed) will be executed at the isolation level value in effect when the cursor was opened. The change will not take effect for operations on that cursor until it has been closed and opened again. Note that the changed isolation level **will** be used (without error) for SQL statements not referencing the opened cursor.

If the program sets the isolation level variable to a value other than 'C' or 'R', or if it fails to initialize the variable at all, SQL/DS will stop execution and return an error code in the SQLCA.

Figure 27 shows the isolation level variable name for each of the four host languages.

Host Language	Variable Name	Example
Assembler	SQLISL	SQLISL DS CL1
COBOL	SQL-ISL	01 SQL-ISL PIC X(1).
FORTRAN	SQLISL	CHARACTER SQLISL

Figure 27 (Part 1 of 2). Variable Names for Specifying Mixed Isolation Levels

Host Language	Variable Name	Example
PL/I	SQLISL	DCL SQLISL CHAR(1);

Figure 27 (Part 2 of 2). Variable Names for Specifying Mixed Isolation Levels

Note: If you forget to declare the isolation level variable in a PL/I program, the PL/I compiler will issue an informational message which might, in some environments, be suppressed. An example that shows how to mix isolation levels in a PL/I program is contained in Appendix C.

Isolation level cursor stability only has meaning for data in PUBLIC DBSPACES with ROW or PAGE level locking. Data in PRIVATE DBSPACES and PUBLIC DBSPACES with DBSPACE level locking always use repeatable read isolation. However, programs which access such data and do not require repeatable read should be preprocessed with cursor stability. The data concurrency requirements might change and cause the data to be moved to a PUBLIC DBSPACE with PAGE or ROW level locking. In this case, the program would not need to be re-preprocessed to run at isolation level cursor stability.

When SQL/DS uses a DBSPACE scan (does not use an index) to access a table in a DBSPACE with ROW level locking using isolation level cursor stability, the effect is the same as repeatable read. That is, no other logical unit of work can update the table until the logical unit of work performing the DBSPACE scan ends. Also, if one logical unit of work has updated a table, another logical unit of work (using cursor stability) cannot access that table with a DBSPACE scan until the updating logical unit of work ends. This reduced concurrency for DBSPACE scans does not apply for tables in DBSPACES with PAGE level locking, or when accessing through indexes. Since most data base accesses will typically use indexes, the reduced concurrency caused by DBSPACE scans should not occur frequently.

Also note that the isolation level specification affects UPDATE and DELETE processing as well as SELECT processing. For UPDATE and DELETE processing, SQL/DS frequently acquires SHARE locks that will be released quickly with cursor stability, but will be held until the end of the LUW with repeatable read. The use of data administration commands such as CREATE, ACQUIRE, or GRANT should not play a role in your choice of isolation level. They use repeatable read locking no matter what the isolation level is set to. Also, catalog access for SQL statement preprocessing is always done with repeatable read locking.

Previous releases of SQL/DS supported only isolation level repeatable read. If you have old programs which might qualify for cursor stability locking, just re-preprocess and recompile these programs with cursor stability specified. You do not need to make any programming changes.

To Block or Not to Block?

SQL/DS gives you the option of telling SQL/DS to insert and retrieve rows in groups or blocks, instead of one at a time. This is called specifying the **blocking** option. When you specify the blocking option, performance improves for SQL/DS application programs that:

1. Execute in multiple user mode, and
2. Retrieve or insert multiple rows.

You can specify the blocking option as an SQL/DS preprocessor parameter, or as an option on the CREATE PROGRAM statement. After an SQL/DS program has been preprocessed with the blocking option, all **eligible** cursor SELECTs and all **eligible** cursor INSERTs within the program will be blocked. You don't have to specify a block size or block factor. SQL/DS automatically fixes the block size for inserts and for SELECTs.

Which programs would benefit from blocking? Programs that do multiple-row inserts (with PUT statements) or multiple-row SELECTs (with FETCH statements) are most likely to perform better when you specify blocking for them. In both of these cases, a cursor must be defined. (See "Retrieving or Inserting Data with a Cursor" on page 19.) Thus, follow this general rule for blocking:

USE BLOCKING FOR PROGRAMS THAT DECLARE CURSORS

A program can use either PUT or FETCH statements without being sensitive to whether SQL/DS will be blocking. That is, PUT and FETCH will work, regardless of whether you specified the blocking option.

If you are moving from an earlier release of SQL/DS, you may wish to re-preprocess existing programs which might benefit from blocking for cursor SELECTs. Just preprocess the program again with the BLOCK option. You don't need to make any changes to the program itself.

Remember that when you preprocess a program with the blocking option, *all* eligible INSERTs and SELECTs are blocked. You cannot specify blocking for just INSERTs or for just SELECTs. If you specify the blocking option, it automatically applies to both.

When are INSERTs or SELECTs not eligible for blocking? SQL/DS sometimes overrides blocking for a particular cursor because of storage limitations in the SQL/DS virtual machine, or because of SQL statement ineligibility. The following SQL statements are ineligible for blocking and cause blocking to be overridden automatically for the cursors they refer to:

- DELETE...WHERE CURRENT OF...
- UPDATE...WHERE CURRENT OF...
- SELECT...FOR UPDATE
- Any SQL statement which contains a long field.

Preparing a SELECT or INSERT statement in single user mode also causes blocking to be overridden. SQL/DS also disqualifies blocking if it cannot fit at least two rows into a block.

SQL/DS does *not* halt the program when it overrides blocking. Instead, in each of the above cases, it sets a warning flag in the SQLCA. The warning can be detected by using WHENEVER SQLWARNING in the program. See “Error Handling” on page 202 for more information on the SQLCA and the SQL WHENEVER declarative statement.

SQL/DS also overrides blocking for all programs running in single user mode. However, in this instance, SQL/DS does *not* return a warning to the SQLCA.

Note: You should *always* CLOSE a cursor before issuing a COMMIT WORK, especially when blocking. If you commit changes before closing a cursor, you will get an error.

Including External Source Files

The inclusion of external files is indicated to the SQL/DS preprocessor by an embedded SQL/DS command, the INCLUDE command, in the user’s source code. It is indicated within the source code where the external source is to be placed. The syntax for the INCLUDE command is as follows:

```
INCLUDE text-name
```

where text-name identifies the external source files. text-name is a one to eight character identifier and cannot be delimited by double quotes. The first character of text-name must be a letter (A-Z), \$, #, or @; the remaining characters must be letters, numbers (0-9), \$, #, @, or underscore (__) unless further restricted by the operating system. Also, text-name cannot be SQLCA or SQLDA, as these are special include keywords.

The statements contained in the external source specified by text-name may be host language statements or SQL/DS statements (except for another INCLUDE command). INCLUDE commands may not be nested, but the external source may contain INCLUDE SQLDA or INCLUDE SQLCA commands. The INCLUDE command may appear in an SQL DECLARE section or the entire SQL DECLARE section(s) may be placed within an external source file(s).

Including Secondary Input

The SQL/DS INCLUDE command may be used to obtain secondary input from a VM/CMS file. If a source program input to an SQL/DS preprocessor uses the INCLUDE facility, any files to be used as secondary input must be accessed by the user. A search of all accessed CMS mini-disks for the filename and filetype is conducted in standard CMS search order (A-Z); the first match determines the filemode. This filename, filetype, and filemode are used as the secondary input or external source. The CMS file containing the secondary input statements must be fixed-length, 80-character records.

The SQL/DS INCLUDE command causes input to be read from the specified filename until the end of the file, at which time the SYSIN input resumes. The file to be included must have an appropriate filetype:

ASMCOPY filetype - Assembler

COBCOPY filetype - COBOL

FORTCOPY filetype - FORTRAN

PLICOPY filetype - PL/I

The filemode is determined by the search of the virtual machine's accessed mini-disks. If the INCLUDE command specifies a file name that is not located on any user-accessed CMS mini-disk, an error will result.

Secondary input must not contain preprocessor INCLUDE commands other than INCLUDE SQLDA or INCLUDE SQLCA, although it may contain both host language and SQL/DS statements. If an INCLUDE command is encountered, an error will result.

In the INCLUDE command, text-name specifies the filename of the secondary input source; the filetype is determined by which preprocessor is invoked. The filemode must be a CMS mini-disk accessed by the user's virtual machine. The text-name must not contain the filetype identifier or filemode.

Chapter 3. SQL Programming Language Reference Summary

This chapter is a “quick” reference and reminder for SQL statements. Each entry contains a brief description of a statement and its parameters. In addition, a chart shows the statement syntax, an example, and authorizations necessary to use the statement.

For the details of how a statement works or peculiarities in how it runs, see the corresponding description of the statement in one of the first two chapters in this book. However, if you have trouble remembering how to code a statement or what its parameters are for, refer to this chapter.

Because Chapter 4, “Extended Dynamic Statements,” is a self-contained chapter, and because it contains its own reference section, the statements introduced in that chapter are not included in this one.

How to Interpret SQL Format

Each SQL statement consists of the statement name followed by one or more keywords (the statement name itself is a keyword). Statement names and keywords may have parameters associated with them. These parameters can be constants or user-defined variables called *host-program variables* (or *host variables* for short). In describing the format of SQL statements in this book, uppercase characters indicate parts that must be coded as shown (statement names and keywords). Lowercase characters indicate that you are to enter a value in their place.

```
CREATE [UNIQUE] INDEX index-name ON table-name
|         |         |         |         |         |-> value you provide
|         |         |         |         |-----> keyword
|         |         |         |-----> value you provide
|         |         |-----> keyword
|         |-----> keyword
|-----> keyword
```

The brackets [] in the above example (a portion of the CREATE INDEX statement) indicate that “UNIQUE” is optional. Do not include the brackets when writing the statement; they serve only as indicators for optional parts of the statement. For example, you could write the above portion of the CREATE INDEX statement as follows:

```
CREATE UNIQUE INDEX FAST1 ON INVENTORY
```

An option that consists of a choice of items (keywords or parameters) has a vertical bar | separating the choices. For example,

```
[AAA|BBB|CCC]
```

indicates that you can specify “AAA” or “BBB” or “CCC” or none; you cannot specify more than one. There can also be options within options, such as:

```
[AAA|BBB[,CCC]|DDD]
```

which means, if you choose “BBB” you can also include “CCC” if you wish. Notice that a comma is included to separate the items if “BBB” and “CCC” are chosen. Commas are used throughout the statement formats in this manner.

To illustrate several items, one, and only one, of which *must* be chosen, braces { } are used. For example,

```
{AAA|BBB|CCC}
```

means you must choose "AAA" or "BBB" or "CCC." When there are many items they are often stacked.

When the items are stacked within braces { }, you *must* choose one:

```
{ AAA } (If you can choose more than one, it is explained  
  BBB ) with the individual statement description.)  
  CCC }
```

When items are stacked within brackets [], you can either choose one of the items or none of the items:

```
[ AAA ] (If you can choose more than one, it is explained  
  BBB ) with the individual statement description.)  
  CCC ]
```

Sometimes a parameter constant is *underlined*. This means it is the default -- the constant used by SQL/DS if none is written. For example,

```
[AAA|BBB|CCC]
```

indicates that if none is chosen, "BBB" is assumed.

To illustrate a long string of items, an ellipsis (...) is used. For example,

```
column-1, column-2, ..., column-n
```

means column-1 to column-n (where n is the last column).

Unlike brackets [] and braces { }, parentheses () are coded as part of the actual statement.

```
RELOAD TABLE([creator.]table-name)
```

In the above example (a portion of the Data Base Services utility RELOAD TABLE statement), the parentheses are part of the statement and must be coded:

```
RELOAD TABLE(JOANNE.EMPTABLE)
```

For easy reading, statement formats in this book are shown spread over several lines, with lines after the first often indented. Statement examples are also shown in this manner. When coding statements in your program, line breaks and indentations are not important unless there are host-language restrictions.

SQL Statement Reference Summary

Contents

ACQUIRE DBSPACE	263
ALTER DBSPACE	265
ALTER TABLE	267
BEGIN DECLARE SECTION	268
CLOSE	269
COMMENT	270
COMMIT WORK	272
CONNECT	273
CREATE INDEX	274
CREATE SYNONYM	276
CREATE TABLE	277
CREATE VIEW	279
DECLARE CURSOR	281
DELETE	284
DESCRIBE	286
DROP DBSPACE	288
DROP INDEX	289
DROP PROGRAM	290
DROP SYNONYM	291
DROP TABLE	292
DROP VIEW	293
END DECLARE SECTION	294
EXECUTE	295
EXECUTE IMMEDIATE	296
EXPLAIN	297
FETCH	299
GRANT	300
INSERT	305
LABEL	309
LOCK	312
OPEN	313
PREPARE	314
PUT	316
REVOKE	317
ROLLBACK WORK	321
SELECT	322
UPDATE	324
UPDATE STATISTICS	328

ACQUIRE DBSPACE

Format:

```
ACQUIRE {PUBLIC | PRIVATE} DBSPACE  
  
    NAMED [owner.]dbspace-name  
  
    ( [ NHEADER = {8|integer}  
      PAGES     = {128|integer}  
      PCTINDEX  = {33|integer}  
      PCTFREE   = {15|integer}  
      LOCK      = {PAGE|size}  
      STORPOOL  = {integer} ] )
```

Note: Any number of options may be specified in any order (separate them with commas).

The LOCK parameter is applicable to PUBLIC DBSPACES only.

Example:

```
ACQUIRE PRIVATE DBSPACE NAMED MFBSpace  
    (STORPOOL=3, PCTFREE=25)
```

Authorization:

You must have DBA authority to acquire either a PUBLIC DBSPACE or a DBSPACE for another user. You must have RESOURCE authority to acquire a PRIVATE DBSPACE.

The ACQUIRE DBSPACE statement causes SQL/DS to find an available DBSPACE of the requested type (PUBLIC or PRIVATE) and give it the dbspace-name you specify.

PUBLIC | PRIVATE

is the type of DBSPACE. If the DBSPACE type is PUBLIC, its owner becomes PUBLIC; if the type is PRIVATE, its owner becomes the user who preprocessed the program in which the ACQUIRE DBSPACE is embedded.

owner.

is the user for whom you are acquiring the DBSPACE, followed by a period. If you are acquiring a DBSPACE for yourself, you need not specify this parameter. If owner is specified when acquiring a PUBLIC DBSPACE, it is ignored.

dbspace-name

is the name you wish to give to the DBSPACE you are acquiring. The name must be an SQL identifier, as described under "General Rules for Naming Data Objects" on page 74. It must be unique within all the DBSPACES

owned by the same user, but may duplicate the name of a DBSPACE owned by another user.

NHEADER

is the number of 4096-byte logical pages in the DBSPACE that SQL/DS reserves for header pages. SQL/DS uses header pages to record information about the contents of the DBSPACE. NHEADER cannot be larger than eight pages, which is the default.

PAGES

is the minimum number of 4096-byte logical pages you require for this DBSPACE. SQL/DS determines the number of pages you receive by rounding the number you specify to the *next higher* multiple of 128 pages. If you do not specify PAGES, SQL/DS acquires the smallest available DBSPACE (128 pages) by default.

PCTINDEX

is the percentage of *all* pages in the DBSPACE that SQL/DS is to reserve for the construction of indexes. If you don't specify PCTINDEX, the default is 33 percent.

PCTFREE

is the percentage of the space on *each* page that SQL/DS is to keep empty when data is inserted into the DBSPACE. If you don't specify PCTFREE, the default is 15 percent.

LOCK

is the lock size, applicable to PUBLIC DBSPACES only. The lock size determines the size of the locks that SQL/DS acquires when a user reads or updates data. The valid specifications for *size* are DBSPACE, PAGE, and ROW. The default LOCK for each PUBLIC DBSPACE is PAGE. If you specify ROW, SQL/DS locks only an individual row in the table; PAGE or DBSPACE cause the smallest lockable unit to be a page (4096 bytes) or a DBSPACE, respectively.

STORPOOL

is the storage pool number. This parameter indicates that SQL/DS must acquire this DBSPACE from the specified storage pool. If a DBSPACE of the specified type and size is not available in this storage pool, the ACQUIRE DBSPACE is not successful; SQL/DS returns a negative SQLCODE. If you don't specify STORPOOL, SQL/DS acquires a DBSPACE of the correct type and size from *any recoverable* storage pool.

ALTER DBSPACE

Format:

```
ALTER DBSPACE [owner.]dbspace-name ( { PCTFREE = integer } )  
                                     { LOCK = size } )
```

Examples:

```
ALTER DBSPACE MFBSpace (PCTFREE = 0)  
ALTER DBSPACE PUBLIC.Space (LOCK = PAGE,PCTFREE=3)
```

Authorization:

To alter a PRIVATE DBSPACE, you must either own the DBSPACE, or have DBA authority. You must have DBA authority to alter a PUBLIC DBSPACE.

With the ALTER DBSPACE statement, you can alter the percentage of free space that SQL/DS reserves on each data page when records are inserted into a PUBLIC or PRIVATE DBSPACE. You can also alter the lock size of a PUBLIC DBSPACE. (You can't alter the lock size of a PRIVATE DBSPACE.)

owner.

is the userid of the owner of the DBSPACE that is to be altered. It is not necessary for DBSPACES that you own. If the DBSPACE is PUBLIC, then the owner is "PUBLIC." To alter another user's DBSPACE, you must have DBA authority.

dbspace-name

is the name of the DBSPACE that you wish to change.

PCTFREE

is a keyword telling SQL/DS that you wish to alter the percentage of space on each data page that is to be kept empty. It is a performance consideration. If an index has been defined for some table in the DBSPACE, the use of reserved free space may result in a more favorable placement of data on pages and, therefore, improve access time.

LOCK

is a keyword telling SQL/DS that you wish to alter the lock size of a public DBSPACE. The valid lock sizes are ROW, PAGE, and DBSPACE. If ROW is specified, the system locks only individual rows in the DBSPACE. PAGE causes the smallest lockable unit to be a page (approximately 4000 characters); DBSPACE causes this unit to be a DBSPACE. The default is PAGE. The LOCK parameter applies to PUBLIC DBSPACES only.

You can specify both the PCTFREE and LOCK parameters when altering a PUBLIC DBSPACE. If you specify both, you can specify them in any order, but you must separate them with a comma. (See the example above.)

ALTER TABLE

Format:

```
ALTER TABLE [creator.]table-name  
  ADD column-name data-type
```

Example:

```
ALTER TABLE SCOTT.SUPPLIERS ADD RATING CHAR(1)
```

Authorization:

To alter a table, you must either be its creator or have the ALTER privilege on it. You can alter any table if you have DBA authority (even the SQL/DS catalogs).

With the ALTER TABLE statement, you can add a single column to an existing table. The new column is added to the right-hand side of the table. All existing rows of the table are expanded and assigned the null value for the new column.

creator.

is the userid of the owner of the table that you wish to add the column to, followed by a period. It is not necessary for tables that you own.

table-name

is the name of the table that you wish to add the column to.

column-name

is the name of the column that you are adding. The new column-name must not be the same as the name of any existing column in the table. Once a column-name is established, it remains for the life of the table. To change it, you must drop the table and re-create it.

data-type

is the data type of the column to be created. For a discussion and listing of valid data types, see "Data Types" on page 75. Any of these data types can be used in the ALTER TABLE statement. However, because the new column is initially assigned null values, you cannot specify the NOT NULL option for the new column.

BEGIN DECLARE SECTION

Format:

```
BEGIN DECLARE SECTION
```

Authorization:

Anyone connected to SQL/DS can issue this statement.

BEGIN DECLARE SECTION is a declarative statement that must be coded in the application prolog. It is used to delineate the beginning of the SQL/DS host variable declaration section. The host variable declaration section is ended by the **END DECLARE SECTION** statement.

CLOSE

Format:

```
CLOSE cursor-name
```

Example:

```
CLOSE C1
```

The **CLOSE** statement is used to stop the usage of the group of rows pointed to by the named cursor (*cursor-name*). This cursor must be in the open state in order to be closed. A cursor is opened using the **OPEN** statement.

When the **CLOSE** statement is executed, the indicated cursor leaves the open state, and its active set becomes undefined. No **FETCH** or **PUT** statement can be executed on the cursor, and no **DELETE** or **UPDATE** statement can refer to its current position, until the cursor is reopened by an **OPEN** statement. **CLOSE** permits SQL/DS to release the resources associated with maintaining an open cursor. **CLOSE** should be placed in your program so that it is executed as soon as the program is finished using a cursor.

When a **CLOSE** statement is executed in a program that is blocking, the remaining rows in an incomplete block are inserted or retrieved.

Note that both the **COMMIT WORK** and **ROLLBACK WORK** statements automatically close all cursors.

COMMENT

Format 1:

```
COMMENT ON { TABLE [creator.]table-name           } IS quoted-string  
           { COLUMN [creator.]table-name.column-name } }
```

Examples:

```
COMMENT ON COLUMN QUOTATIONS.DELIVERY_TIME IS 'MEASURED IN DAYS'  
COMMENT ON TABLE INVENTORY IS 'INVENTORY OF MACHINE PARTS ONLY'
```

Authorization:

You may use Format 1 of the COMMENT statement only if you are the creator of the table in question or if you have DBA authority.

The SQL COMMENT statement lets you associate an explanatory comment with a table or view, or with a column in a table or view. SQL/DS places this comment into the REMARKS column of either the SYSCATALOG or the SYSCOLUMNS catalog table. When you do a SELECT on either of these catalog tables, you can see all the comments you put on your tables or columns.

TABLE

is a keyword telling SQL/DS that you are putting a comment on a table or a view, as opposed to a column. The comment that you specify will be put into the REMARKS column of the SYSTEM.SYSCATALOG catalog table, on the row for the table you are commenting on.

COLUMN

tells SQL/DS that you are commenting on a column of a table or view. The comment that you specify will be put into the REMARKS column of the SYSTEM.SYSCOLUMNS catalog table, on the row for the column of the table or view you are commenting on.

creator.

is the userid of the owner of the table or view which you are commenting on. It is not necessary for tables that you own. To comment on another user's table, you must have DBA authority.

table-name

is the name of the table or view that you wish to comment on. If you are commenting on a column, it is the name of the table or view that contains the column.

column-name
is the name of the column that you wish to comment on.

quoted-string
is the actual comment that you wish to have placed in the appropriate SQL/DS catalog. The comment must not exceed 254 characters, and must be enclosed in single quotes, as shown above. If there already is a comment there, the new comment replaces the old one.

Format 2:

```
COMMENT ON [creator.]table-name (column-name IS quoted-string,  
.....  
column-name IS quoted-string)
```

Example:

```
COMMENT ON QUOTATIONS (SUPPNO IS 'SEE SUPPLIERS TABLE FOR NAMES',  
PRICE IS 'IN U.S. DOLLARS', DELIVERY_TIME IS 'MEASURED IN DAYS')
```

Authorization:

You may use Format 2 of the COMMENT statement only if you are the creator of the table in question or if you have DBA authority.

Format 2 of the SQL COMMENT statement lets you specify comments for more than one column in a table at one time. It can be used only for columns, not for tables or views. SQL/DS places the comments into the REMARKS column of the SYSCOLUMNS catalog table, on the rows associated with the columns that you are commenting on.

creator.
is the userid of the owner of the table or view containing the columns that you are commenting on. It is not necessary for tables that you own. To comment on another user's table, you must have DBA authority.

table-name
is the name of the table or view that contains the columns you wish to comment on.

column-name
is the name of a column that you wish to comment on.

quoted-string
is the actual comment that you wish to have placed in the SYSCOLUMNS catalog. The comment must not exceed 254 characters, and must be enclosed in single quotes, as shown above. If there already is a comment there, the new comment replaces the old one.

COMMIT WORK

Format:

```
COMMIT WORK [RELEASE]
```

Authorization:

Anyone connected to SQL/DS can issue this statement.

The COMMIT WORK statement ends the current logical unit of work if one is in progress and commits any changes made during the logical unit of work to the data base. It is strongly recommended that each application program explicitly end its logical unit of work before terminating.

RELEASE

specifies to SQL/DS that when the COMMIT work processing is done, your connection to SQL/DS is ended. See "COMMIT WORK" on page 233 for more information on this parameter.

CONNECT

Format:

```
CONNECT userid  
IDENTIFIED BY password
```

Example:

```
CONNECT CRAIG  
IDENTIFIED BY :SECRET
```

Authorization:

You must have CONNECT authority to issue this statement.

The CONNECT statement connects an application program to the SQL/DS data base so it can access and work with SQL/DS data. The CONNECT statement also identifies the user to SQL/DS, and determines whether the user has the proper authorization to preprocess or run the program containing the CONNECT statement. See "Connecting to SQL/DS" on page 91 for more information.

The CONNECT statement is not required because SQL/DS does implicit connecting (from VM/SP userids and passwords) if an explicit CONNECT is not found.

If you do choose to code a CONNECT statement, it must be the first SQL statement executed in your program. However, SQL declarative statements (such as the host variable declaration section and the SQLCA structure) may precede the CONNECT statement. The CONNECT statement ends the application prolog.

userid

is the userid of the person trying to CONNECT to SQL/DS. It must be declared as a fixed-length character string of length 8 and must be initialized before the CONNECT statement is executed. However, a userid can be less than 8 characters long. Unused character positions in the host variable for the userid are padded with blanks.

password

is the password of the person trying to CONNECT to SQL/DS. It must be a host variable. It must be declared as a fixed-length character string of length 8 and must also be initialized before the CONNECT statement is executed. And, a password can be less than 8 characters long.

CREATE INDEX

Format:

```
CREATE [UNIQUE] INDEX index-name ON [creator.]table-name
  ( column-name-1 [ ASC | DESC ]
    [,column-name-2 [ ASC | DESC ] ] ... )
  [PCTFREE= {10|integer} ]
```

Example:

```
CREATE INDEX FASTQUOTES ON QUOTATIONS
  (PARTNO ASC, PRICE DESC, DELIVERY_TIME)
  PCTFREE = 33
```

Authorization:

You can create indexes on tables that you have created. You need the INDEX privilege or DBA authority to create an index on another user's table (for example, CREATE INDEX MYID.IND1 ON OTHERID.TAB1). You need DBA authority to create an index *for* another user (for example, CREATE INDEX OTHERID.IND1 ON OTHERID.TAB1).

The CREATE INDEX statement creates an index on one or more columns of a table and gives a name to the new index. The SQL/DS preprocessor chooses which index, if any, is to be used in processing a given query or data manipulation statement. The index provides SQL/DS with a fast means to access the table directly by the indexed column. However, there is a slight increase in the time required to update the indexed column because SQL/DS must also update the index.

When you preprocess a program, SQL/DS creates an access module for it that takes advantage of the best access path available at the time of the preprocessing. Therefore, it is good practice to create indexes *before preprocessing* programs that might take advantage of them. When you create a new index, existing access modules are not made invalid because they can still use their original access path. However, an existing program may run more efficiently by taking advantage of the new index. If this is the case, you should preprocess the program again. A new access module is then created for the program, possibly using the new index.

An index is maintained by SQL/DS until it is explicitly dropped in a DROP INDEX statement, or until its table or DBSPACE is dropped. Indexes cannot be created for views or for columns whose data type is LONG VARCHAR or LONG VARGRAPHIC.

UNIQUE

ensures that no two rows of the indicated table are identical in the indexed column(s). If, when a CREATE UNIQUE INDEX statement is executed, the table already contains some rows that are not unique in the indexed

columns, the CREATE UNIQUE INDEX statement fails and returns an error code. You can force all rows of a table to be unique by creating a unique index on all columns of the table.

index-name

is the name you wish to give to the index you are creating.

creator.

is the userid of the owner of the table that you wish to put the index on, followed by a period. It is not necessary for tables that you own.

table-name

is the name of the table that you wish to create the index on.

column-name-1

is the name of the primary column that you wish to create the index on.

column-name-2, column-name-3, ...

are the names of additional columns that you wish to create the index on. See "Creating an Index" on page 242 for a list of rules to follow when creating an index on multiple columns.

ASC | DESC

specifies the order of the index on a column. You can create an index on a column in either ascending (ASC) or descending (DESC) order. Ascending order is the default. Performance may be improved for queries that access the indexed column in the specified order.

PCTFREE

controls the amount of free space reserved in an index for later insertions and updates. PCTFREE defines the percentage (integer) of the total space of the index that is to be reserved for this purpose. It may range from 0 to 99, but for practical purposes should not exceed 50. Increasing PCTFREE causes the index to take more space in the data base, but reduces the time required to insert or update rows of the indexed table. If you don't include a PCTFREE clause in the CREATE INDEX statement, SQL/DS sets PCTFREE to the default value of 10.

CREATE SYNONYM

Format:

```
CREATE SYNONYM identifier FOR creator. {table-name | view-name}
```

Example:

```
CREATE SYNONYM PARTS FOR SMITH.INVENTORY
```

Authorization:

Anyone connected to SQL/DS can issue this statement. You can create a synonym for any user's table or view. (No authorization is required.)

The CREATE SYNONYM statement defines an alternative name for a table or view. This allows you to refer to a table or view owned by you or another user without having to enter the fully-qualified name. Once created, a synonym remains in effect until it is explicitly dropped by a DROP SYNONYM statement.

identifier

is the synonym (alternative name) you wish to use to refer to the table or view. Rules for identifiers are outlined under "General Rules for Naming Data Objects" on page 74.

creator.

is the userid of the owner of the table or view that you wish to create the synonym for, followed by a period. This is required, even when you are creating a synonym for one of your own tables or views.

table-name | view-name

is the name of the table or view for which you wish to create the synonym.

CREATE TABLE

Format:

```
CREATE TABLE [creator.]table-name
  (column-name-1 data-type-1 [NOT NULL]
  [,column-name-2 data-type-2 [NOT NULL] ] ... )
  [IN [owner.]dbspace-name]
```

Example:

```
CREATE TABLE MIKE.MUSICIANS
  ( NAME          VARCHAR(20) NOT NULL,
    INSTRUMENT    VARCHAR(10) NOT NULL,
    "BAND NAME"   VARCHAR(10)      )
  IN MIKE.DBSP1
```

Authorization:

You must have **RESOURCE** authority to create a table, unless someone with **DBA** authority has acquired a **PRIVATE DBSPACE** on your behalf. If a **DBA** does acquire a **DBSPACE** for you, you may create a table in that **DBSPACE**, even if you do not have **RESOURCE** authority. You must have **DBA** authority to create a table for another user.

The **CREATE TABLE** statement creates a new table in the data base with the table name and the column names that you specify. Once a table has been created, you may not change the data types of its columns or drop a column from the table. However, you may add new columns to the table by the **ALTER TABLE** statement.

creator.

is the userid of the person who you are creating the table for, followed by a period. When you create tables for yourself, you do not need to specify this parameter. If you are creating a table in a program and you do not specify creator, creator defaults to the user who preprocesses the program.

table-name

is the name that you wish to give the table you are creating. The table-name must follow the rules for an SQL identifier as discussed under "General Rules for Naming Data Objects" on page 74.

column-name-1, column-name-2, ...

are the names you wish to give to the columns of the table. Column names must also follow the rules for an SQL identifier.

data-type-1, data-type-2, ...

are the types of data (such as integer, decimal or character) that you want each column to have. Valid data types for columns are listed under "Data Types" on page 75. Any of those data types can be used as data types in SQL/DS tables.

NOT NULL

is a parameter telling SQL/DS not to permit null values in that column. Any statement that attempts to place a null value in such a column is rejected with an error code.

owner.

is the owner of the DBSPACE that you are creating the table in.

dbspace-name

is the name of the DBSPACE that you are creating the table in. You don't have to specify this (or owner.) if you are creating the table in one of your own DBSPACES. You can avoid confusion, however, by specifying who you are creating the table for (creator.) and where you want the table to be placed (owner.dbspace-name).

CREATE VIEW

Format:

```
CREATE VIEW [creator.]view-name [(column-name-list)]
  AS select-statement
```

Example:

```
CREATE VIEW FASTQUOTES (MFR,PART,DAYS) AS
  SELECT SUPPNO, PARTNO, DELIVERY_TIME
  FROM QUOTATIONS WHERE DELIVERY_TIME < 10
```

Authorization:

You must have the **SELECT** privilege on the underlying tables to create a view.

The **CREATE VIEW** statement causes the indicated select-statement to be stored as the definition of a new view. The statement also gives a name to the view, and (optionally) to each column in the view. Host variables are not allowed in a **CREATE VIEW** statement.

creator.

is the userid of the person who you are creating the view for. This defaults to the person issuing the command in ISQL, or to the person preprocessing the program that the **CREATE VIEW** statement is in.

view-name

is the name that you wish to give the view that you are creating. It must follow the rules for identifiers as outlined under “General Rules for Naming Data Objects” on page 74. Also, the name of the view must be unique among all the tables, views, and synonyms that you have already created.

column-name-list

is the list of the names that you wish to give particular columns in the view you are creating. If you don't specify the column names, the columns of the view inherit the names of the columns from which they are derived. You must specify new names for virtual columns. See “Creating a View” on page 140 for the definition of a virtual column. You must also specify new column names if the selected fields of the view do not have unique names (for example, the view is a join of two tables, each of which has a column named **PARTNO**).

Internal SQL/DS limitations restrict a view to approximately 140 columns. The number of referenced tables, lengths of column names, and **WHERE** clauses all further reduce this number.

select-statement

is the **SELECT** statement that defines the view you are creating. It is coded in basic form; that is, with just the **SELECT**, **FROM** and **WHERE** clauses included, and no cursor defined.

If you use a "**SELECT ***" clause in the view definition and you later add columns to the underlying table with **ALTER** statements, the new columns will *not* appear in the view. Therefore, you may wish to avoid this type of clause.

The select-statement must not have an **ORDER BY** clause and cannot contain a **UNION** operator.

The select-statements that define the various views known to the **SQL/DS** are kept in a catalog called **SYSVIEWS**. Also, descriptions of views and their columns are kept in **SYSCATALOG** and **SYSCOLUMNS**.

DECLARE CURSOR

Format 1:

```
DECLARE cursor-name CURSOR FOR select-statement  
    [ ORDER BY o-spec [ASC|DESC] [, o-spec [ASC|DESC] ] ...  
    [ FOR UPDATE OF column-name-1 [, column-name-2 ] ... ]
```

Example:

```
DECLARE C1 CURSOR FOR SELECT PARTNO, PRICE  
    FROM QUOTATIONS WHERE SUPPNO=:SUPP  
    ORDER BY PARTNO
```

Authorization:

Anyone connected to SQL/DS can issue this statement. You must, however, be authorized to access the tables referenced in the SELECT statement. (See the SELECT statement authorization description.)

Format 1 of the DECLARE CURSOR statement should not be confused with a host variable declaration. This statement defines a cursor by associating a *cursor-name* with the specified *select-statement*. The set of rows defined by the specified select-statement are called the active set of the cursor. Once the cursor is defined, you can manipulate the active set using the OPEN, FETCH and CLOSE statements.

Note: The DECLARE CURSOR statement for dynamically defined queries is not reviewed in this chapter. It is discussed in detail under "Dynamically Defined Statements" on page 147.

cursor-name

is the name you wish to give to the cursor you are defining. Cursor names must be unique in a logical unit of work. They must follow all the usual rules for SQL identifiers and, unlike other SQL identifiers, cursor names must never be enclosed in either single (') or double (") quotes. Thus, cursor names cannot contain embedded blanks. Cursor names can, however, be SQL reserved words.

select-statement

defines the active set of the cursor. All the rows that satisfy the search conditions in this select-statement become part of the active set, which you can manipulate using OPEN, FETCH, and CLOSE statements.

ORDER BY

causes SQL/DS to deliver the rows of the active set in the specified order. This clause is discussed in detail under "More About Cursor Management" on page 134.

o-spec

is the order specification for the active set. It is a list of column names or integers that refer to select-list items for the purpose of indicating an ordering in the ORDER BY clause.

ASC | DESC

indicates the order that you want a column ordered by: either ascending or descending order. Ascending is the default.

FOR UPDATE OF

is an optional clause telling SQL/DS that you might want to update some columns of the active set. This clause is discussed in more detail under "More About Cursor Management" on page 134.

column-name-1, column-name-2, ...

are the names of columns that you might want to update via this cursor. You can only update these columns listed in the FOR UPDATE clause.

Format 2:

```
DECLARE cursor-name CURSOR FOR insert-statement
```

Examples:

```
DECLARE C2 CURSOR FOR INSERT INTO INVENTORY  
  (PARTNO, DESCRIPTION, QONHAND)  
  VALUES (:PART, :DESC, :QUAN)
```

```
DECLARE C3 CURSOR FOR INSERT INTO QUOTATIONS  
  (SUPPNO, PARTNO, QONORDER)  
  VALUES (58, 207, 11)
```

Authorization:

Anyone connected to SQL/DS can issue this statement. You must, however, be authorized to access the tables referenced in the INSERT statement. (See the INSERT statement authorization description.)

Format 2 of the DECLARE CURSOR statement should not be confused with a host variable declaration. This statement defines a cursor for inserting rows into a table by associating a *cursor-name* with the specified *insert-statement*. The set of rows defined by the specified insert-statement are called the **active set** of the cursor. Once the cursor is defined, you can manipulate the active set using the OPEN, PUT and CLOSE statements.

Note: The DECLARE CURSOR statement for dynamically defined inserts is not reviewed in this chapter. It is discussed in detail under "Dynamically Defined Statements" on page 147.

cursor-name

is the name you wish to give to the cursor you are defining. Cursor names must be unique in a logical unit of work. They must follow all the usual rules

for SQL identifiers and, unlike other SQL identifiers, cursor names must never be enclosed in either single (') or double (") quotes. Thus, cursor names cannot contain embedded blanks. Cursor names can, however, be SQL reserved words.

insert-statement

defines the active set of the cursor. You can manipulate this active set with the OPEN, PUT, and CLOSE statements. This active set consists of one row of data which is to be inserted into an SQL/DS table. The list-of-data-items in the insert-statement (usually a list of host variables) defines the values that you wish to insert by using the PUT statement.

DELETE

Format 1 DELETE:

```
DELETE FROM [creator.]table-name  
    [WHERE search-condition]
```

Examples:

```
DELETE FROM QUOTATIONS WHERE SUPPNO = 53  
DELETE FROM QUOTATIONS WHERE DELIVERY_TIME IS NULL  
DELETE FROM QUOTATIONS WHERE PARTNO = :X AND PRICE > :Y  
DELETE FROM SCOTT.INVENTORY WHERE DESCRIPTION = 'PISTON'
```

Authorization:

You can delete rows from any table you create. You can delete rows from another user's table if you are given the DELETE privilege on that table, or if you have DBA authority.

Format 1 of the DELETE statement deletes one or more rows from a given table. SQL/DS deletes all rows of the named table that satisfy the search condition.

creator.

is the userid of the owner of the table that you wish to delete rows from. If you do not specify this parameter, it defaults to the person who preprocessed the program that the DELETE statement is contained in.

table-name

is the name of the table that you wish to delete rows from.

search-condition

describes the row or rows to be deleted. The search condition may take any of the forms described under "Using Expressions as Search Conditions" on page 30.

If you omit the search condition, SQL/DS deletes all rows from the named table, but sets a warning indicator in SQLWARN4. You can check SQLWARN4 to detect unintentional deletions and rollback the logical unit of work before the changes are permanently committed to the data base.

If no rows satisfy the given search condition, SQL/DS returns the "not found" code (SQLCODE=100).

Format 2 DELETE:	
	DELETE FROM [creator.] table-name WHERE CURRENT OF cursor-name
Example:	
	DELETE FROM INVENTORY WHERE CURRENT OF CURSOR2
Authorization:	
	Authorization depends on the table specified in the cursor declaration. You can delete rows of the table named in the cursor declaration if you created that table. If you are not the creator of the table in the cursor declaration, you must be given the DELETE privilege on that table or you must have DBA authority.

Format 2 of the DELETE statement deletes exactly one row of a table. The current position of the cursor determines the row to be deleted.

creator.

is the userid of the owner of the table that you wish to delete a row from. If you do not specify this parameter, it defaults to the person who preprocessed the program that the DELETE statement is contained in.

table-name

is the name of the table that you wish to delete a row from.

cursor-name

is the name of the cursor that is pointing to the row that you wish to delete. The cursor must be open and positioned on a row of the table. Also, it must be defined by a SELECT statement on one table (not a join). If the SELECT statement contains a subquery, the subquery must not be on the same table as the outer-level query.

DESCRIBE

<p><i>Format:</i></p> <pre>DESCRIBE statement-name INTO output-spec [USING {<u>NAMES</u> LABELS BOTH ANY}]</pre>
<p><i>Examples:</i></p> <pre>DESCRIBE Q1 INTO SQLDA DESCRIBE S1 INTO STR1 USING LABELS DESCRIBE STMT INTO SQLDA USING ANY</pre>
<p><i>Authorization:</i></p> <p>You can use DESCRIBE for any statement you have successfully prepared.</p>

The DESCRIBE statement obtains information about a statement that has been prepared. If the prepared statement is a SELECT statement, DESCRIBE returns the number of fields in the answer set, and the data types, lengths, and names of these fields. If the prepared statement is not a SELECT statement, DESCRIBE sets the SQLDA field called SQLD to zero.

All fields in the SQLDA were described under "The SQL Descriptor Area (SQLDA)" on page 167. General usage techniques are described under "Dynamically Defined Queries" on page 151.

You should not attempt to DESCRIBE a statement that was prepared in a different logical unit of work. If you do, the results are unpredictable.

statement-name

is the name of the statement that you are preparing.

output-spec

should name an SQLDA structure.

NAMES

tells SQL/DS to return column names but no column labels to the SQLNAME fields of SQLDA. This is the default.

LABELS

tells SQL/DS to return column labels but no column names to the SQLNAME fields of SQLDA.

BOTH

tells SQL/DS to return both column labels and column names to the SQLNAME fields of the SQLVAR array in SQLDA. The value returned in SQLDA is twice the number of columns (N) in the select-list.

ANY

If a label exists for a column, it is returned to the **SQLNAME** field. If not, the column name is returned.

DROP DBSPACE

Format:

```
DROP DBSPACE [owner.]dbspace-name
```

Examples:

```
DROP DBSPACE MFBSpace  
DROP DBSPACE MIKE.MFBSpace  
DROP DBSPACE PUBLIC.SPAC10
```

Authorization:

A DBSPACE may be dropped only by its owner or by a user having DBA authority. You must have DBA authority to drop a PUBLIC DBSPACE. No user, even with DBA authority, can drop the DBSPACE containing the SQL/DS catalogs.

The DROP DBSPACE statement destroys the contents of a DBSPACE and returns the DBSPACE to an “available” state. All existing access modules for programs that operate on the dropped DBSPACE are automatically marked “invalid.” You can use DROP DBSPACE with both PUBLIC and PRIVATE DBSPACES.

owner.

is the owner of the DBSPACE that you wish to drop.

dbspace-name

is the name of the DBSPACE that you wish to drop.

DROP INDEX

Format:

```
DROP INDEX [creator.]index-name
```

Example:

```
DROP INDEX FASTQUOTES
```

Authorization:

You can use the DROP INDEX statement only if you are the creator of the index to be dropped (or if you have DBA authority).

The DROP INDEX statement drops the indicated index from the data base. The table on which the index is defined is not affected. All existing access modules that use the dropped index are marked invalid.

creator.

is the owner of the index that you wish to drop.

index-name

is the name of the index that you wish to drop.

DROP PROGRAM

Format:

```
DROP PROGRAM [creator.]program-name
```

Examples:

```
DROP PROGRAM PAYROLL2  
DROP PROGRAM SALLY.RUNRUN  
DROP PROGRAM :CREATOR.:PROGNAME
```

Authorization:

You can only drop programs that you have preprocessed. (That is, you must be the creator of the program you wish to drop.) To drop another user's program, you must have DBA authority.

The **DROP PROGRAM** statement erases the access module associated with the named program. Once you drop an access module, you cannot run the program.

creator.

is the owner of the program that you wish to drop.

program-name

is the name of the program that you wish to drop. This is the name specified in the **PREPNAME** parameter when the program is preprocessed.

DROP SYNONYM

Format:

```
DROP SYNONYM identifier
```

Example:

```
DROP SYNONYM DATA
```

Authorization:

You can drop a synonym only if you have created it.

With the **DROP SYNONYM** statement you can erase a synonym from the data base. Then you can no longer use this synonym as an alternate name for a table unless you re-create it using the **CREATE SYNONYM** statement. The table on which the synonym is based is not affected.

identifier

is the synonym (referring to a table or view) that you wish to drop.

DROP TABLE

Format:

```
DROP TABLE [creator.]table-name
```

Example:

```
DROP TABLE INVENTORY
```

Authorization:

You can drop a table only if you have created the table, or if you have DBA authority.

This statement drops the indicated table from the data base. All indexes and views defined on the table, and all privileges granted on the table, are also dropped from the data base. All contents of the table are lost.

creator.

is the owner of the table that you wish to drop. It is not necessary for tables that you own.

table-name

is the name of the table that you wish to drop.

DROP VIEW

Format:

```
DROP VIEW [creator.]view-name
```

Example:

```
DROP VIEW FASTQUOTES
```

Authorization:

You can drop only those views that you have created. You can drop another user's views only if you have DBA authority.

The **DROP VIEW** statement drops the definition of the indicated view from the data base.

creator.

is the owner of the view that you wish to drop.

view-name

is the name of the view that you wish to drop.

END DECLARE SECTION

Format:

```
END DECLARE SECTION
```

Authorization:

Anyone connected to SQL/DS can issue this statement.

END DECLARE SECTION is a declarative statement that must be coded in the application prolog. It is used to delineate the end of the SQL/DS host variable declaration section. The host variable declaration section is begun by the BEGIN DECLARE SECTION statement.

EXECUTE

Format 1:

```
EXECUTE statement-name [USING input-list]
```

Format 2:

```
EXECUTE statement-name [USING DESCRIPTOR input-structure]
```

Examples:

```
EXECUTE S1 USING :X, :Y:YIND  
EXECUTE S1 USING DESCRIPTOR SQLDA  
EXECUTE S1 USING DESCRIPTOR STUFF
```

Authorization:

Any user with **CONNECT** authority can code this statement in an application program and preprocess the program. SQL statements submitted to SQL/DS via the **PREPARE** and **EXECUTE** facility have their authorization checked against the privileges of the user who is currently running the program, *not* the user who preprocessed the program.

Format 1 of the **EXECUTE** statement causes SQL/DS to execute a statement that was “prepared” previously. When the statement is executed, the host variables you list are substituted, in order, into the statement in place of its “?” parameters.

Format 1 of the **EXECUTE** statement is used when you know the number and data types of the parameters of the prepared statement. Format 2 permits you to dynamically specify the “?” parameters of the prepared statement. If you use Format 2, you must use the **SQLDA** to specify the required parameters. General usage techniques for the **SQLDA** were discussed under “Dynamically Defined Queries” on page 151.

statement-name

is the name of the statement that you wish to execute. This statement must have been prepared earlier by a **PREPARE** statement.

input-list

is the list of host variables that you wish to substitute, in order, into the prepared statement in place of its “?” parameters.

input-structure

is the **SQLDA** that specifies information for each variable represented by a “?” in the prepared statement. See “Dynamically Defined Queries” on page 151 for more information.

EXECUTE IMMEDIATE

Format:

```
EXECUTE IMMEDIATE string-spec
```

Example:

```
EXECUTE IMMEDIATE :QSTRING
```

Authorization:

Any user with CONNECT authority can code this statement in an application program and preprocess the program. SQL statements submitted to SQL/DS via EXECUTE IMMEDIATE have their authorization checked against the privileges of the user who is currently running the program, *not* the user who preprocessed the program.

The EXECUTE IMMEDIATE statement is a short-hand form of the PREPARE and EXECUTE statements. It is used for preparing and executing SQL statements having no parameters. EXECUTE IMMEDIATE should be used when the SQL statement is to be executed only once. If a given SQL statement is to be prepared once and executed repeatedly, the non-immediate form of EXECUTE should be used.

string-spec

identifies the statement that you want to prepare and execute. (See "PREPARE" on page 314 for string-spec syntax rules.)

EXPLAIN

Format:

```
EXPLAIN explain-spec [SET QUERYNO = small-integer-value] FOR sql-command
```

Example:

```
EXPLAIN REFERENCE, STRUCTURE, COST, PLAN
SET QUERYNO = 1500
FOR SELECT * FROM QUOTATIONS
WHERE SUPPNO >= 53
```

Authorization:

You must own an explanation table for each of the specified explain-spec options. Also, you must have the proper privileges to execute the SQL statement defined by sql-command.

The EXPLAIN statement retrieves information about the structure and execution performance of an SQL command. It places the information into one or more SQL/DS explanation tables.

The result tables built by the EXPLAIN statement are created during preprocessing of the containing program. You can also PREPARE/EXECUTE or EXECUTE IMMEDIATE an EXPLAIN statement.

explain-spec

is the name of the explanation table(s) into which information is to be placed. explain-spec may include one or more of the following options, separated by commas:

REFERENCE for information contained in the REFERENCE__TABLE

STRUCTURE for information contained in the STRUCTURE__TABLE

COST for information contained in the COST__TABLE

PLAN for information contained in the PLAN__TABLE

ALL for information contained in all of the above tables.

small-integer-value

is an integer constant that can fit into a SMALLINT field. The SET QUERYNO clause allows you to place an integer value into the QUERYNO fields of the rows in the explanation tables. Assigning a different number on each EXPLAIN will make it easier to identify information collected.

sql-command

is the SQL command to be analyzed. You can analyze UPDATE, DELETE, and INSERT commands as well as SELECT commands. (SELECT commands are considered the primary candidates for EXPLAIN analysis). sql-command is not a quoted string and must not be put in a host variable.

The length of the SQL statement is limited to about 8000 characters.

FETCH

Format:

```
FETCH cursor-name INTO host-list
```

Exa

```
FETCH C1 INTO :NAME, :ADDR, :PHONE:PHONI
```

The FETCH statement retrieves a row from the active set into specified host variables. The position of the cursor is advanced to the next row of the active set, and the selected fields of this row are delivered into the output host variables specified in the host-list.

Note: The FETCH statement for dynamically defined queries is not reviewed in this chapter. It is discussed in detail under “Dynamically Defined Statements” on page 147. In FORTRAN programs, only the simple form of the FETCH statement (as listed in this section) is allowed.

cursor-name

is the name of the cursor defining the active set that you wish to retrieve a row from. The cursor must have been defined in terms of a SELECT statement and must be in the open state.

host-list

is the list of output host variables that you want the retrieved values delivered into. Output host variables in this list must be separated by commas, and must be immediately preceded by colons.

GRANT

Format 1 (for privileges on tables and views):

```
GRANT { [ ALTER
        DELETED
        INDEX
        INSERT
        SELECT
        UPDATE [(col-name-list)]
        ALL [PRIVILEGES] ] }

ON [creator.] {table-name | view-name}
TO { PUBLIC | userid1 [,userid2] ... } [WITH GRANT OPTION]
```

Note: ALTER, INDEX, and ALL [PRIVILEGES] do not apply to views.

Examples:

```
GRANT UPDATE (PARTNO, SUPPNO) ON QUOTATIONS TO SCOTT
GRANT SELECT, INSERT ON QUOTATIONS TO SMITH, JONES
GRANT ALL PRIVILEGES ON INVENTORY TO SCOTT WITH GRANT OPTION
```

Authorization:

You must possess the privilege with the GRANT option before you can grant that privilege to someone else.

With Format 1 of the GRANT statement, you can grant privileges on tables and views to other users. This is usually issued through ISQL or the DBS utility.

You can specify more than one privilege. If you do, you can specify them in any order, but you must separate them with commas.

Note that only the user who creates a table or view (or a user with DBA authority) can drop it. You can't grant a "drop" privilege to another user.

ALTER

enables the specified users to add new columns to the specified table. This privilege does not apply to views or DBSPACES. (That is, it applies to the ALTER TABLE statement, but not the ALTER DBSPACE statement.)

DELETE

enables the specified grantee(s) to delete rows from the table or view.

INDEX

allows the grantees to create indexes on the specified table. This privilege does not apply to views.

INSERT

lets the grantees insert rows into the table or view.

SELECT

enables the grantees to insert rows into the table or view.

UPDATE

allows the grantees to update the table or view.

col-name-list

gives the grantee the power to update only the columns listed. If you choose not to specify a list of column names or if you specify ALL [PRIVILEGES], the grantee may update all columns of the table, even those created later via the ALTER TABLE statement.

ALL [PRIVILEGES]

grants all six privileges to the specified user(s). This parameter lets you write ALL [PRIVILEGES] instead of listing all six. (Note that you can't grant ALL PRIVILEGES on a view; INDEX and ALTER privileges do not apply to views.) The PRIVILEGES keyword is both optional and non-functional; you can include it to improve readability.

creator.

is the userid of the owner of the table or view that you wish to grant privileges on. This is not necessary for tables that you own.

table-name

is the name of the table that you wish to grant privileges on.

view-name

is the name of the view that you wish to grant privileges on.

PUBLIC

grants the specified privilege(s) to all users.

userid1, userid2, ...

are the userid(s) of the user(s) to whom you wish to grant the specified privileges.

WITH GRANT OPTION

enables the grantee to pass the granted privileges to other users.

Format 2 (for privileges on programs):

```
GRANT RUN ON [creator.]program-name  
TO { PUBLIC | userid1 [,userid2] ... } [WITH GRANT OPTION]
```

Examples:

```
GRANT RUN ON TRANS1 TO EDWARDS WITH GRANT OPTION  
GRANT RUN ON JOB338 TO PUBLIC
```

Authorization:

You must possess the RUN privilege with the GRANT option before you can grant that privilege to someone else.

Format 2 allows you to grant privileges on programs to other users. The only privilege you can grant on a program is the RUN privilege, which lets another user run the indicated program. This is usually issued through ISQL or the DBS utility.

Note that only the user who preprocesses a program (or a user with DBA authority) can drop its access module from the data base. You can't grant a "drop" privilege to another user.

creator.

is the userid of the person who preprocessed the program that you wish to grant privileges on. This is not necessary for programs that you own.

program-name

is the name of the program that you wish to grant privileges on.

PUBLIC

grants the RUN privilege to all users.

userid1, userid2, ...

are the userid(s) of the user(s) to whom you wish to grant the RUN privilege.

WITH GRANT OPTION

enables the grantee to pass the RUN privilege to other users.

Format 3 (for special privileges):

```
GRANT { CONNECT
      { DBA
      { RESOURCE
      { SCHEDULE } } } TO userid1[,userid2...] [IDENTIFIED BY pass1[,pass2]]
```

Examples:

```
GRANT DBA TO BRUCE
GRANT CONNECT TO SMITH, JONES IDENTIFIED BY SECRET1, SECRET2
GRANT RESOURCE TO MARY, JIM, JOE
```

Authorization:

Generally, you must possess DBA authority to issue this statement. The exception is that can change your own password as explained below.

Format 3 allows a *user having DBA authority* to grant special privileges to other users. The special privileges are CONNECT, DBA, RESOURCE and SCHEDULE authority.

CONNECT

tells SQL/DS to grant CONNECT authority to the specified user(s). A user can use this parameter with the IDENTIFIED BY clause to change his/her own password. Granting CONNECT to ALLUSERS is a special case that establishes implicit connect capability for all users in the system when operating under VM/SP.

DBA

tells SQL/DS to grant DBA authority to the specified user(s). This also means that the specified user(s) will be automatically granted CONNECT authority.

RESOURCE

tells SQL/DS to grant RESOURCE authority to the specified user(s). This also means that the specified user(s) will be automatically granted CONNECT authority.

SCHEDULE

tells SQL/DS to grant SCHEDULE authority to the specified user(s). This also means that the specified user(s) will be automatically granted CONNECT authority. Note that a grant of SCHEDULE authority to a user is meaningless because SQL/DS allows only resource managers to use it.

userid1, userid2, ...

are the user(s) to whom you wish to grant special privileges. Userids are limited to eight characters.

IDENTIFIED BY

adds or changes the password for each user specified. If you specify IDENTIFIED BY, you must include a password for every userid specified.

pass1, pass2, ...

specify the new or changed password(s) for each of the specified user(s). Passwords are limited to eight characters. The passwords and userids must correspond as indicated in the statement format above. If the password is the same as currently exists for the user, or if no passwords are specified, the change has no real effect.

INSERT

Format 1 INSERT:

```
INSERT INTO [creator.]table-name [(list-of-column-names)]
VALUES (list-of-data-items)
```

Examples:

```
INSERT INTO JONES.INVENTORY (PARTNO,DESCRIPTION,QONHAND)
VALUES (251,'GEAR',:QOH:IND1)
INSERT INTO QUOTATIONS VALUES (:A,:B,:C:CI,:D:DI,:E:EI)
INSERT INTO QUOTATIONS VALUES (68,209,18.00,14,0)
INSERT INTO WEATHER (DATE, LOCATION, TEMPERATURE)
VALUES ('JANUARY 13, 1981','ENDICOTT',-15)
```

Authorization:

You can insert data into any table you create. You can insert data into another user's table if you are given the INSERT privilege on that table, or if you have DBA authority.

Format 1 of the INSERT statement inserts a single new row into an existing table. SQL/DS forms the new row by placing the various data-items into the specified columns in the order named. Rows are inserted in SQL/DS-determined order. That is, no facility is provided to specify the "position" in the table of the newly inserted rows.

For inserting more than one row into a table, you may wish to use the SQL PUT statement (see "PUT Statement" on page 25).

creator.

is the userid of the owner of the table that you wish to insert a row into. This must be specified for tables owned by another user (as shown in the first example above). It is not necessary to specify for tables that you own.

table-name

is the name of the table that you wish to add a row to. This must follow the rules for SQL identifiers as outlined under "General Rules for Naming Data Objects" on page 74.

list-of-column-names

is the list of columns in the specified table that you wish to insert values into, separated by commas. All columns of the table that you do not name receive the null value. You do not have to list the column names in the same sequence that they were named when the table was created. Omitting the list of column names is the same as naming all the columns in the order that they were named when the table was created.

list-of-data-items

is the list of pieces of data to be inserted, separated by commas. Data items can be numeric or alphabetic constants. They can also be host variables, with or without associated indicator variables. The NULL keyword can also be used as a data item, indicating to SQL/DS that you wish to insert a null value into the table.

The data types of the values to be inserted (source data type) do not necessarily have to match the data types defined for the columns (target data type). However, the data types must be compatible, that is, character to character, numeric to numeric, or DBCS to DBCS. SQL/DS performs data conversion automatically on compatible data types. Data conversion rules are summarized under "Data Conversion" on page 76.

Format 2 INSERT:

```
INSERT INTO [creator.]table-name [(list-of-column-names)]
select-statement
```

Example:

```
INSERT INTO MYPARTS
  SELECT PARTNO, DESCRIPTION, PRICE
  FROM SCOTT.PARTS
  WHERE DESCRIPTION = 'PISTON'
```

Authorization:

You can insert data into any table you create. You can insert data into another user's table if you are given the INSERT privilege on that table, or if you have DBA authority. You must have proper SELECT authorization on those tables referenced in the select-statement.

Format 2 of the INSERT statement inserts into an existing table one or more rows. These rows are selected or computed from other tables by a SELECT statement. If SQL/DS detects an error in a Format 2 INSERT statement after some rows have been inserted, SQL/DS stops processing the statement and returns an error code in the SQLCA.

Rows are inserted in SQL/DS-determined order. That is, no facility is provided to specify the "position" in the table of the newly inserted rows.

creator.

is the userid of the owner of the table that you wish to insert rows into. This must be specified for tables owned by another user. It is not necessary for tables that you own.

table-name

is the name of the table that you wish to insert rows into. This must follow the rules for SQL identifiers as outlined under "General Rules for Naming Data Objects" on page 74.

list-of-column-names

is the list of the columns in the specified table that you wish to insert values into, separated by commas. All columns in the table that you do not name in the list-of-column-names receive the null value in each row inserted. You do not have to list the column names in the same sequence that they were named when the table was created. Omitting the list of column names is the same as naming all the columns in the order that they were named when the table was created.

select-statement

is the SELECT statement you are using to indicate which rows (from other tables) you wish to insert into the specified table (table-name). A SELECT

statement used in an INSERT must be in “basic” form; that is, it must not have an INTO clause.

If the number of columns selected by the SELECT statement is not equal to the number of columns needed for the insertion, an error results. Also, the nested SELECT statement must not select rows from the same table that is the subject of the INSERT, since this might lead to a non-terminating result.

The data types of the values to be inserted (source data type) do not necessarily have to match the data types defined for the columns (target data type). However, the data types must be compatible, that is, character to character, numeric to numeric, or DBCS to DBCS. SQL/DS performs data conversion automatically on compatible data types.

You cannot use Format 2 of the INSERT statement to insert data of type LONG VARCHAR or LONG VARGRAPHIC.

LABEL

Format 1:

```
LABEL ON { TABLE [creator.]table-name  
          COLUMN [creator.]table-name.column-name } IS quoted-string
```

Examples:

```
LABEL ON TABLE QUOTATIONS IS 'CURRENT PRICE QUOTATIONS'  
LABEL ON COLUMN INVENTORY.PARTNO IS 'PART NUMBER'
```

Authorization:

Only the creator of the table or a user with DBA authority can issue the LABEL statement on a table or column.

The SQL LABEL statement lets you define a label for a table name or column name. Labels are used as common presentation headings. You can specify a query using column names and table names (easier to enter) and see the data with labels (easier to understand). Using labels provides you with the same headings on all the reports created by various applications that access your labeled tables.

SQL/DS stores the labels in either the TLABEL column of the SYSCATALOG catalog table or in the CLABEL column of the SYSCOLUMNS catalog table. SQL/DS returns the labels to the application program via the SQLDA structure. The program requests the labels by specifying one of the following options on the DESCRIBE statement: LABELS, ANY or BOTH. See "DESCRIBE" on page 286 for more information on the DESCRIBE statement.

TABLE

is a keyword telling SQL/DS that you are defining a label for a table or a view name, as opposed to for a column name. The label that you define will be stored in the TLABEL column of the SYSTEM.SYSCATALOG catalog table, on the row for the table you are labeling.

COLUMN

tells SQL/DS that you are defining a label for a column of a table or view. The label that you define will be stored in the CLABEL column of the SYSTEM.SYSCOLUMNS catalog table, on the row for the column of the table or view you are defining the label for. Column labels are returned in the SQLDA when a "SELECT" statement is described by a "DESCRIBE" statement.

creator.

is the userid of the owner of the table or view which you are defining the label for. It is not necessary for tables that you own. To define a label for another user's table, you must have DBA authority.

table-name

is the name of the table or view that you wish to define a label for. If you are defining a label for a column, it is the name of the table or view that contains the column.

column-name

is the name of the column that you wish to define a label for.

quoted-string

is the actual label that you wish to define for your table or column. The maximum length of a label is 30 bytes. The label must be enclosed in single quotes, as shown above. If a label already exists for the table or column that you specify, the new label replaces the old one.

Format 2:

```
LABEL ON [creator.]table-name (column-name IS quoted-string,
                               .....
                               column-name IS quoted-string)
```

Example:

```
LABEL ON INVENTORY (PARTNO IS 'PART NUMBER', DESCRIPTION IS
'PART DESCRIPTION', QONHAND IS 'QUANTITY ON HAND')
```

Authorization:

Only the creator of the table or a user with DBA authority can issue the LABEL statement on a column.

Format 2 of the SQL LABEL statement lets you create labels for more than one column in a table. It can be used only for columns, not for tables or views. SQL/DS stores the labels in the CLABEL column of the SYSCOLUMNS catalog table. SQL/DS returns the labels to the application program via the SQLDA structure. The program requests the labels by specifying one of the following options on the DESCRIBE statement: LABELS, ANY or BOTH. See "DESCRIBE" on page 286 for more information on the DESCRIBE statement.

creator.

is the userid of the owner of the table or view which contains the columns that you are defining the labels for. It is not necessary for tables that you own. To define a label on someone else's table, you must have DBA authority.

table-name

is the name of the table or view which contains the columns that you wish to define the labels for.

column-name

is the name of a column that you wish to define a label for.

quoted-string

is the actual label that you wish to define for a column. The maximum length of a label is 30 bytes. The label must be enclosed in single quotes, as shown above. If a label already exists for a column that you specify, the new label replaces the old one.

LOCK

Format:

```
LOCK {TABLE [creator.]table-name|DBSPACE [owner.]dbspace-name}  
    IN {SHARE|EXCLUSIVE} MODE
```

Examples:

```
LOCK TABLE PARTS IN EXCLUSIVE MODE  
LOCK DBSPACE DSP3 IN SHARE MODE
```

Authorization:

To lock a DBSPACE, you must either be the owner of the DBSPACE, or have DBA authority. You can lock a table if you own the table, if you have the SELECT privilege on the table, or if you have DBA authority. No user, regardless of authority, can lock any SQL/DS catalog or the DBSPACE containing the catalogs.

The LOCK statement overrides the SQL/DS automatic locking mechanism. It explicitly acquires a lock on a table or DBSPACE. SQL/DS holds the requested lock until the end of the current logical unit of work.

A LOCK statement on a table in a PRIVATE DBSPACE is the same as a LOCK statement on the entire DBSPACE, since locking is always done at the DBSPACE level for PRIVATE DBSPACES.

TABLE

indicates to SQL/DS that you want to acquire a lock on the named table (table-name).

creator.

is the owner of the table that you wish to acquire the lock on.

DBSPACE

indicates to SQL/DS that you want to acquire a lock on the named DBSPACE (dbspace-name).

owner.

is the owner of the DBSPACE that you wish to acquire the lock on.

SHARE | EXCLUSIVE

indicates to SQL/DS the mode of the lock you wish to acquire. An *exclusive* lock prevents other users from reading or changing any data in the locked table or DBSPACE. A *share* lock permits other users to read, but prevents them from modifying, the data in the locked object.

OPEN

Format:

```
OPEN cursor-name
```

Examples:

```
OPEN C1
```

If you are opening a query-cursor, the OPEN statement examines the input host variables (if any) used in the definition of the named cursor (cursor-name), determines the active set for the cursor, and leaves it in the open state. When SQL/DS executes an OPEN statement for a query-cursor, it positions the cursor *before* the first row of the active set. No rows in the active set are actually fetched to the host program until a FETCH statement is executed.

If you are opening an insert-cursor and your program is blocking, this statement tells SQL/DS to prepare to block the rows to be inserted. If you are not blocking, SQL/DS prepares to insert a single row into the data base. Rows are not actually inserted into the data base until one or more PUT statements have been executed.

Note: The OPEN statement for dynamically defined queries and inserts is not reviewed in this chapter. It is discussed in detail under “Dynamically Defined Statements” on page 147. In FORTRAN programs, only the simple form of the OPEN statement (as listed in this section) is allowed.

PREPARE

Format:

```
PREPARE statement-name FROM string-spec
```

Examples:

```
PREPARE STAT2 FROM :XSTRING  
PREPARE STAT3 FROM 'DELETE FROM QUOTATIONS WHERE PARTNO = ?'
```

Authorization:

Any user with CONNECT authority can code this statement in an application program and preprocess the program. SQL statements submitted to SQL/DS via the PREPARE and EXECUTE facility have their authorization checked against the privileges of the user who is currently running the program, *not* the user who preprocesses the program.

This statement preprocesses the statement identified by *string-spec* for later execution. The “prepared” statement is given the *statement-name* you specify.

statement-name

is the name that you wish to give to the statement being “prepared.” It must follow the rules for SQL identifiers as discussed under “General Rules for Naming Data Objects” on page 74. Unlike other SQL identifiers, however, the statement-name must never be enclosed in either single (') or double (") quotes; thus, the statement-name cannot contain embedded blanks. Statement-names can, however, be SQL reserved words.

string-spec

identifies the run-time SQL statement that you wish to prepare. String-spec can be either a character constant or a host variable. If string-spec is a host variable, the variable must be declared as fixed- or varying-length character. Assembler language programs cannot specify a constant for string-spec. A host variable *must* be used, and it *must* be varying length. Also, if a host variable is used in COBOL, it must be varying-length.

The SQL statements you cannot use for string-spec are:

INCLUDE SQLCA	ROLLBACK WORK
INCLUDE SQLDA	COMMIT WORK
WHenever	CONNECT
OPEN	PREPARE
CLOSE	EXECUTE
FETCH	EXECUTE IMMEDIATE
DECLARE CURSOR	DESCRIBE

The SQL statements must not include host language delimiters or contain any references to host variables. If the SQL statement is a **SELECT** statement, it must not have an **INTO** clause.

A question mark (representing a parameter to be filled in when the statement is executed) can appear in an SQL statement to be “prepared” in any place that a host variable may appear, with the following exceptions:

1. A question mark cannot be used in a select-list or **FROM**-clause (but it may be used in the **WHERE** clause of a **SELECT** statement).
2. Two question marks cannot appear directly within the same arithmetic or comparison operation: $?+?$ or $?=?$ are invalid.
3. A question mark cannot be the first item in an **IN**-predicate.

PUT

Format:

```
PUT cursor-name
```

Example:

```
PUT C1
```

The PUT statement is most often used to insert rows into SQL/DS tables in groups or blocks. Each time a PUT statement is executed, a single row of data is added to the insert-block. Rows are not inserted into the data base until the block is full, or until a CLOSE statement is executed. The PUT statement can also be executed when blocking is not in effect. In this case, one row of data is inserted directly into an SQL/DS table. The PUT statement can only be executed when the indicated cursor is in the OPEN state.

Note: The PUT statement for dynamically defined inserts is not reviewed in this chapter. It is discussed in detail under “Dynamically Defined Statements” on page 147. In FORTRAN programs, only the simple form of the PUT statement (as listed in this section) is allowed.

cursor-name

is the name of the cursor defining the row that you wish to insert. You must define this cursor with an SQL DECLARE CURSOR statement, in terms of a Format 1 INSERT statement. (Format 2 is acceptable, but relatively useless.) The DECLARE CURSOR statement should contain a DECLARE clause (defining the name of the cursor), an INSERT clause (indicating which table and which columns you wish to insert the row into), and a VALUES clause (defining the values to be inserted). The values to be inserted can be defined through input host variables or constants. See “Retrieving or Inserting Data with a Cursor” on page 19 for more information on the DECLARE CURSOR statement.

REVOKE

Format 1 (for privileges on tables and views):

```
REVOKE { [ ALTER  
         DELETE  
         INDEX  
         INSERT  
         SELECT  
         UPDATE  
         ALL [PRIVILEGES] ] } ON [creator.] {table-name | view-name}  
FROM { PUBLIC | userid1 [,userid2] ... }
```

Note: ALTER, and INDEX, do not apply to views. ALL [PRIVILEGES] does apply, however. (See following text.)

Examples:

```
REVOKE SELECT, INSERT ON QUOTATIONS FROM SMITH, JONES  
REVOKE UPDATE ON INVENTORY FROM PUBLIC  
REVOKE ALL ON SUPPLIERS FROM SCOTT
```

Authorization:

You can revoke only those privileges you have granted to other users, not those another user has granted.

With Format 1 of the REVOKE statement, you can revoke privileges you have granted on tables and views. The specified privileges on the specified table or view are revoked from the specified user(s). This is usually issued through ISQL or the DBS utility.

You can specify more than one privilege that you wish to revoke. If you do, you can specify them in any order, but you must separate them with commas.

Note that the only way to revoke the GRANT option on a privilege is to revoke the privilege itself.

ALTER

tells SQL/DS to take away the privilege to add new columns.

DELETE

tells SQL/DS to revoke the privilege to delete rows.

INDEX

tells SQL/DS to take away the privilege to create indexes.

INSERT

tells SQL/DS to take away the privilege to insert rows.

SELECT

tells SQL/DS to revoke the privilege to insert rows.

UPDATE

tells SQL/DS to revoke the privilege to update any columns.

ALL [PRIVILEGES]

tells SQL/DS to take away all the table (or view) privileges granted by you to the specified user(s). This parameter lets you write ALL [PRIVILEGES] instead of listing all the privileges you granted. You can use this parameter even if you did not grant all six table privileges to the user. The PRIVILEGES keyword is both optional and non-functional; you can include it to improve readability.

creator.

is the userid of the owner of the table or view that you wish to revoke privileges from. This is not necessary for tables that you own.

table-name

is the name of the table that you wish to revoke privileges from.

view-name

is the name of the view that you wish to revoke privileges from.

PUBLIC

tells SQL/DS to revoke the indicated privileges you have explicitly granted to PUBLIC (via GRANT...TO PUBLIC). It does *not* revoke all your grants of the indicated privilege.

userid1, userid2, ...

are the userid(s) of the user(s) from whom you wish to revoke the specified privileges.

Format 2 (for privileges on programs):

```
REVOKE RUN ON [creator.]program-name FROM { PUBLIC | userid1 [,userid2] ... }
```

Example:

```
REVOKE RUN ON TRANS1 FROM SMITH
```

Authorization:

You can revoke the RUN privilege from only those users to whom you have granted it.

Format 2 of the REVOKE statement revokes the RUN privilege you have granted on programs. This is usually issued through ISQL or the DBS utility.

If you have granted the RUN privilege with the GRANT option, the only way to revoke the GRANT option is to revoke the RUN privilege itself.

creator.

is the userid of the owner of the program that you wish to revoke RUN authority from. This is not necessary to specify for programs that you preprocessed.

program-name

is the name of the program that you wish to revoke RUN authority from.

PUBLIC

tells SQL/DS to revoke RUN authority on a program that you have explicitly granted to PUBLIC (via GRANT RUN TO PUBLIC). It does *not* revoke all your grants of the RUN privilege.

userid1, userid2, ...

are the userid(s) of the user(s) from whom you wish to revoke the RUN privilege.

Format 3 (for special privileges):

```
REVOKE {CONNECT | RESOURCE | SCHEDULE | DBA} FROM userid1 [,userid2] ...
```

Example:

```
REVOKE DBA FROM SMITH
```

Authorization:

You must possess DBA authority to issue this statement.

Format 3 of the REVOKE statement allows a *user having DBA authority* to revoke special privileges from other users. This is usually issued through ISQL or the DBS utility.

CONNECT

tells SQL/DS to revoke CONNECT authority from the specified user(s).

Revoking CONNECT causes all special privileges to be revoked with it and the user is deleted from the SQL/DS catalog SYSUSERAUTH.

RESOURCE

tells SQL/DS to revoke RESOURCE authority from the specified user(s).

No one can revoke RESOURCE authority from a user that has DBA authority. Revoking RESOURCE authority implies no other revocations.

SCHEDULE

tells SQL/DS to revoke SCHEDULE authority from the specified user(s).

DBA

tells SQL/DS to revoke DBA authority from the specified user(s). A user having DBA authority cannot revoke any authority from himself. Revoking DBA authority automatically causes all special privileges to be revoked except CONNECT.

userid1, userid2, ...

are the userid(s) of the user(s) from whom you wish to revoke the special privileges.

ROLLBACK WORK

Format:

ROLLBACK WORK [RELEASE]

Authorization:

Anyone connected to SQL/DS can issue this statement.

The ROLLBACK WORK statement ends the current logical unit of work if one is in progress and undoes any changes made during that time. It restores the data base to its state prior to the current logical unit of work. It is strongly recommended that each application program explicitly end its logical unit of work before terminating.

RELEASE

tells SQL/DS that when the ROLLBACK WORK processing is done, your connection to SQL/DS is ended. See "ROLLBACK WORK" on page 234 for more information on this parameter.

SELECT

Format:

```
SELECT [ALL | DISTINCT] { select-list | * }  
INTO one-or-more-host-variables  
FROM table-name  
[ WHERE search-condition ]
```

Example:

```
SELECT PARTNO, QONHAND  
INTO :PARTNO, :QUANT  
FROM INVENTORY  
WHERE DESCRIPTION = 'BOLT'
```

Authorization:

You can select information from tables that you have created. You can select information from another user's table only if you have the SELECT privilege on that table or if you have DBA authority.

The SELECT statement (with the INTO clause) selects one row of data from the specified SQL/DS table(s) and returns the specified column(s) of that row into one or more host variables. The statement first finds the row of the table specified in the in the FROM clause that satisfies the given search condition specified in the WHERE clause. From this row SQL/DS selects the columns that you have supplied in the select-list. The results are delivered into the host variables that you have listed in the INTO clause.

To retrieve more than one row of a table, use Format 1 of the DECLARE CURSOR statement instead. If you are coding in FORTRAN, you must use the DECLARE CURSOR statement, regardless of whether one row or more than one row is to be returned.

ALL

tells SQL/DS that all rows satisfying the search-condition are to be returned, including duplicates. This isn't very useful in the SELECT / INTO statement when only one row can be returned. It is more often used in the select-statement part of the DECLARE CURSOR statement.

DISTINCT

tells SQL/DS that only unique rows satisfying the search-condition are to be returned. That is, duplicate rows are not selected. It can also be used to eliminate duplicates from a built-in function. However, it can only be used once per SELECT statement.

This parameter is more often used in the select-statement part of the DECLARE CURSOR statement.

select-list

is a definition of the data that you want returned into the host variables. It consists of one or more column names or expressions, separated by commas. An expression can be a constant, a host variable, or an arithmetic combination of column names and/or host variables and/or constants. The items in the select-list are retrieved in the same left-to-right order as they appear in the select-list. An item in a select-list may also be preceded by a built-in function.

*

indicates that the data in all the columns of the table is to be selected and placed into host variables. This is the same as specifying all the columns of the table in the same left-to-right order as they are defined in the table.

one-or-more-host-variables

is the list of host variables that are to receive the results of the SELECT statement. After SQL/DS has determined which row is to be returned, SQL/DS delivers into these host variables the field values that you specify in the SELECT clause. The number of host variables that you specify in this list must be equal to the number of expressions or column names that you specify in the select-list. Otherwise, SQL/DS returns an error message.

creator.

is the owner of the table that you wish to select the row from.

table-name

is the name of the table that you wish to select the row from.

search-condition

identifies the row that is to be selected. It consists of one or more conditions that apply to selecting data. See “WHERE Clause: Searching on Conditions” on page 19, “Host Variables and Constants” on page 28, and “Using Expressions as Search Conditions” on page 30 for more information on search conditions.

UPDATE

Format 1 UPDATE:

```
UPDATE [creator.]table-name
SET column-name-1 = expression-1
[, column-name-2 = expression-2] ...
[ WHERE search-condition ]
```

Examples:

```
UPDATE EMPLOYEES
SET SALARY = 65000.00,
    POSITION = 'RETIRED'
WHERE NAME = 'J. B. ROBINSON'
```

```
UPDATE SUPPLIERS
SET NAME = :NAM:INAM,
    ADDRESS = :ADDR:IADDR
WHERE SUPPNO = :SNO
```

Authorization:

You can update tables you create. You can update columns in other user's tables if you are given the UPDATE privilege on the columns, or if you have DBA authority.

A Format 1 UPDATE statement changes the values of one or more fields in one or more rows of a table. SQL/DS updates all rows that satisfy the search condition.

creator.

is the userid of the owner of the table that you wish to update, followed by a period. It is not necessary to specify this parameter when updating tables that you own.

table-name

is the name of the table that you wish to update.

column-name-1, column-name-2, ...

are the names of the columns that you wish to update.

expression-1, expression-2, ...

define values that you want the specified columns set to. Only the rows which satisfy the search condition are updated. SQL/DS then changes the specified columns of these rows, replacing their values with new values defined by the expressions. An expression, as described under "Using Expressions as Search Conditions" on page 30, can contain constants, host variables, field names, and the arithmetic operators +, -, *, and /. An expression can also be the NULL keyword, which sets a column to the null

value. A field name occurring on the right side of an “=” sign in a SET clause represents the value of that field before the update occurs. All the updates specified in an UPDATE statement are done simultaneously, after all the update values have been computed.

Note that SQL/DS applies data conversion to SET clause expressions as described under “Using Expressions as Search Conditions” on page 30.

search-condition

defines the row or rows to be updated. The search condition may take any of the forms described under “Using Expressions as Search Conditions” on page 30.

If you omit the search condition, SQL/DS updates all rows of the named table, but sets a warning indicator in SQLWARN4 so that you can detect unintentional updates and rollback the logical unit of work before the changes are permanently committed to the data base.

If no rows satisfy the search condition, the “not found” code (SQLCODE=100) is returned.

Format 2 UPDATE:

```
UPDATE [creator.]table-name
SET column-name-1 = expression-1
[, column-name-2 = expression-2] ...
WHERE CURRENT OF cursor-name
```

Example:

```
UPDATE JONES.EMPLOYEE
SET SALARY = 0.00,
    POSITION = 'FIRED'
WHERE CURRENT OF CURSOR1
```

Authorization:

Authorization depends on the table specified in the cursor declaration. You can update rows of the table named in the cursor declaration if you created the named table. If you are not the creator of the table in the cursor declaration, you must be given the UPDATE privilege on those columns you wish to update, or you must have DBA authority.

Format 2 of the UPDATE statement changes the values of one or more fields in exactly one row of a table -- the current row of the indicated cursor. The UPDATE statement does not affect the position of the cursor.

creator.

is the userid of the owner of the table that you wish to update, followed by a period. It is not necessary to specify this parameter when updating tables that you own.

table-name

is the name of the table that you wish to update.

column-name-1, column-name-2, ...

are the names of the columns that you wish to update.

expression-1, expression-2, ...

define values that you want the specified columns set to. SQL/DS changes the specified columns of the row pointed to by the cursor, replacing their values with new values defined by the expressions. An expression, as described under "Using Expressions as Search Conditions" on page 30, can contain constants, host variables, field names, and the arithmetic operators +, -, *, and /. An expression can also be the NULL keyword, which sets a column to the null value. A field name occurring on the right side of an "=" sign in a SET clause represents the value of that field before the update occurs. All the updates specified in an UPDATE statement are done simultaneously, after all the update values have been computed.

Note that SQL/DS applies data conversion to SET clause expressions as described under "Using Expressions as Search Conditions" on page 30.

cursor-name

is the name of the cursor pointing to the row that you wish to update. The cursor must be open and positioned on a row of the named table. It must have been defined as a **SELECT** statement on one table (not an **INSERT** or a join).

If the **SELECT** statement which defines the cursor contains a subquery, the subquery must not be on the same table as the outer-level query. Also, this **SELECT** statement must not include **DISTINCT** or **GROUP BY** or **ORDER BY** or **UNION** or any built-in function such as **AVG(PRICE)**. Each field to be updated must have been included in a **FOR UPDATE** clause in the **DECLARE CURSOR** statement that defined the cursor.

UPDATE STATISTICS

Format:

```
UPDATE [ALL] STATISTICS FOR  
  {TABLE [creator.]table-name | DBSPACE [creator.]dbspace-name}
```

Example:

```
UPDATE STATISTICS FOR TABLE QUOTATIONS
```

Authorization:

No authorization is required for this statement. Any user can issue UPDATE STATISTICS for any table or DBSPACE.

You can use the UPDATE STATISTICS statement to bring up to date the internal statistics recorded by SQL/DS for tables and indexes. Invoking UPDATE STATISTICS can improve performance on statements which access data from those tables. These statistics, which are contained in the SQL/DS catalogs, include the size of the table, various index characteristics, and other information. See "Updating Internal Statistics" on page 236 for information on the use of UPDATE STATISTICS.

ALL

tells SQL/DS to update statistics for **all** columns, including those that contain indexes. In the case of columns without indexes, the column statistics are an approximation. If ALL is not specified, statistics are updated for only the columns that contain indexes.

TABLE

indicates to SQL/DS that you wish to update statistics on only the named table (table-name).

creator.

is the userid of the owner of the table or DBSPACE that you wish to update statistics on.

DBSPACE

indicates to SQL/DS that you wish to update statistics on every table in the designated DBSPACE (dbspace-name).

WHENEVER

Formats:

```
WHENEVER SQLERROR {STOP | CONTINUE | {GO TO|GOTO} statement-label}
WHENEVER SQLWARNING {STOP | CONTINUE | {GO TO|GOTO} statement-label}
WHENEVER NOT FOUND {CONTINUE | {GO TO|GOTO} statement-label}
```

Note: The STOP condition is not valid for FORTRAN applications.

Examples:

```
WHENEVER SQLERROR GOTO ERRORX
WHENEVER SQLWARNING CONTINUE
WHENEVER NOT FOUND CONTINUE
```

Authorization:

Anyone connected to SQL/DS can issue this statement.

The **WHENEVER** statement lets you specify an action to be taken depending on what SQL/DS returns in the SQLCA. The **WHENEVER** statement is declarative; it is not executed at run-time and returns no SQLCODE.

The keywords **SQLERROR**, **SQLWARNING**, and **NOT FOUND** in the statement syntax above identify some SQLCA condition. The braced keywords (**STOP**, **CONTINUE**, **GOTO**) define the action to be taken whenever the specified SQLCA condition occurs.

The scope of a **WHENEVER** statement is determined by its position in the source program listing, not by its placement in the logic flow. That is, the scope of the **WHENEVER** is independent of the *execution sequence* of statements.

SQLERROR

is the keyword identifying the SQLCA **SQLERROR** condition. The **SQLERROR** condition exists when SQL/DS has set SQLCODE to a negative value.

SQLWARNING

is the keyword identifying the SQLCA **SQLWARNING** condition. The **SQLWARNING** condition exists when SQL/DS sets SQLWARN0 to 'W'.

NOT FOUND

is the keyword identifying the SQLCA **NOT FOUND** condition. The **NOT FOUND** condition exists when SQLCODE is set to 100.

STOP

causes program termination. If a logical unit of work is in progress, it is rolled back. Note that you can't specify STOP for WHENEVER NOT FOUND or in FORTRAN applications.

CONTINUE

causes the next sequential program instruction to be executed. (The SQLCA condition is ignored.)

GO TO (or GOTO) statement-label

causes control to pass to the statement at the specified label (statement-label). The statement label cannot exceed 18 characters unless the host language has additional limitations.

Chapter 4. Extended Dynamic Statements

Contents

Purpose and Use of Extended Dynamic Statements	332
An Example of Extended Dynamic Statements	336
Logical Unit of Work Considerations	344
Virtual Storage Considerations in a Logical Unit of Work	347
Extended Dynamic Statement Descriptions	348
CREATE PROGRAM	348
Extended PREPARE	350
Extended DESCRIBE	352
Extended EXECUTE	354
Extended DECLARE CURSOR	356
Extended OPEN	358
Extended FETCH	359
Extended PUT	360
Extended CLOSE	361
DROP STATEMENT	362

Purpose and Use of Extended Dynamic Statements

An understanding of dynamically defined statements, presented under “Coding the Program” in Chapter 2, is a prerequisite for this topic.

Extended dynamic statements support direct creation and maintenance of access modules for SQL/DS data. These statements provide a function similar to that provided by the SQL/DS preprocessors, but the functions may be particularly useful where:

- The current preprocessors do not support the language of the application or support program that is needed.
- SQL statements are conceived and built dynamically, but are executed repetitively (in a different logical unit of work). In this case it is a performance benefit to avoid having to repeat the preprocessing of statements each time they are executed, as would be required for normal dynamic statements.
- It is desirable to build and maintain an application package of SQL statements to be shared by a group of users.

Individual SQL statements can be added or deleted without affecting or repeating the preprocessing of other SQL statements in the group (a group can be stored in one access module).

By using the extended dynamic statements, development programmers can write their own preprocessors or data base interface routines that support compiled access to SQL/DS. Compiled access means that access paths to SQL/DS data are optimized once when the statement is prepared. They need not be prepared again for each execution.

Following are some of the areas that may be appropriate for this support:

- Preprocessed stored SQL statements in SQL/DS access modules
- General high-level language support for SQL data base access
- Preprocessor functions in other systems that may support SQL/DS.

The extended dynamic statement support is intended primarily for language and program development use, not directly for application programmers. However, because the extended dynamic statements are available to any program that is written in Assembler, or that can interface with an Assembler module, their use in application programs is not precluded. One example of such an application is one where program storage is critical and there are a significant number of predefined “transactions” involving SQL/DS data. By using extended dynamic statements, the number of host language statements generated by SQL/DS preprocessing of the application may be significantly reduced. All SQL/DS access statements may be prepared using a single PREPARE statement, and all data may be retrieved using a single cursor. Thus an application requiring dozens of SQL/DS statements, with their individual expansions for SQL/DS access, may be handled via a single set of seven or eight extended dynamic statements, without losing the performance

benefits of compiled access. This may result in significant reduction in the size of the application program.

The following extended dynamic statements are supported:

- CREATE PROGRAM - build an empty access module.
- PREPARE - add a statement to an access module.
- DESCRIBE - obtain information about columns in the select list of a prepared SELECT statement.
- EXECUTE - execute a non-SELECT statement in an access module created via CREATE PROGRAM.
- DECLARE CURSOR - in connection with OPEN, FETCH, PUT, and CLOSE, execute a SELECT or an INSERT statement in an access module created via CREATE PROGRAM.
- OPEN (cursor)
- FETCH (cursor)
- PUT (cursor)
- CLOSE (cursor)
- DROP STATEMENT - delete a statement from an access module.

Except for CREATE PROGRAM and DROP STATEMENT, the names of these statements are the same as the corresponding “normal” dynamic statements discussed under “Dynamically Defined Statements” on page 147, but their format and meaning are somewhat different. For example, the statement-id, program-name, and cursor-name fields can all be specified via host variables. Each of these extended dynamic statements is described at the end of this appendix.

Unlike the dynamic statements that are related through a specific statement name, the extended dynamic statements are related through the symbolic host variables used for statement-id and program-name. This relationship between the statements is shown in Figure 28. Because the statement-id and program-name are host variables, actual values can be substituted when the program is executed. STMTID is returned by Extended PREPARE and is used as input by the subsequent Extended EXECUTE (or DECLARE CURSOR) statement.

page 73. Like the extended dynamic statements, `DROP PROGRAM` permits the program name to be specified as a host program variable.

Extended dynamic statements, like all other SQL statements, require preprocessing, but extended dynamic statements are supported by only the Assembler preprocessor. Once they are preprocessed (and the containing program is compiled), the program holding the extended dynamic statements may itself be used to process SQL statements and create access modules. That is, it may prepare SQL statements for repetitive execution.

An Example of Extended Dynamic Statements

Consider the following example. A support group needs to develop a support program that dynamically accepts SQL statements for execution and it does not know what specific SQL statements will be processed. This is a typical application for normal dynamic SQL statements. But if additionally, there is a requirement for repetitively executing the preprocessed statements at a later time (stored SQL application) without having to repeat the `PREPARE`, it is an application for extended dynamic statements.

The support program that supports preparation of end user SQL statements may also support their execution (via commands of the support program). This is essentially a query language support program (but it supports more than just queries (`SELECT` statements)). The support program may also support deleting statements from and adding statements to existing access modules. There are extended dynamic statements for adding statements to, and deleting statements from, an access module:

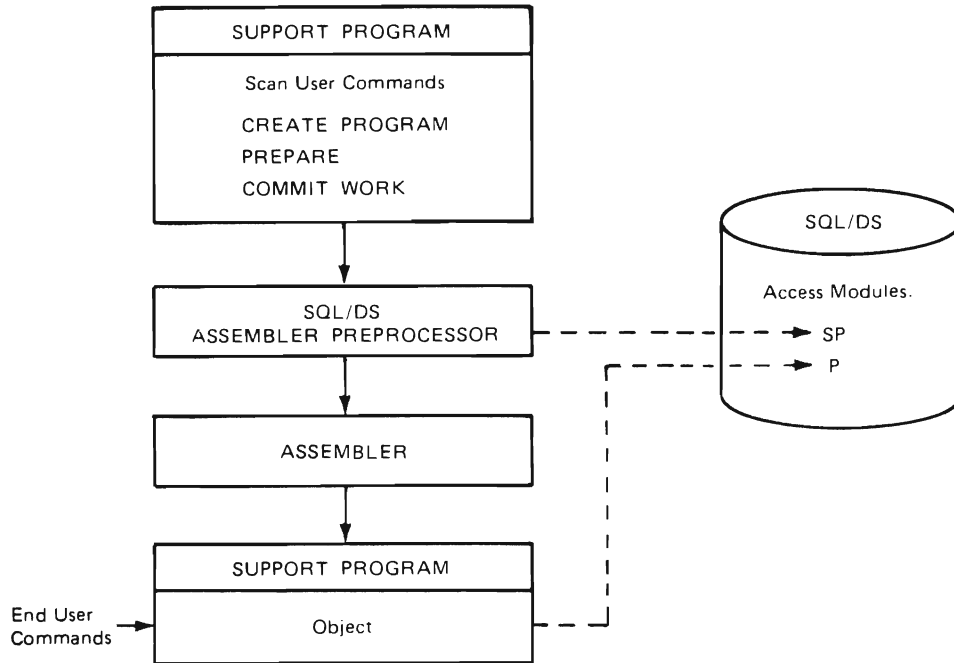
- `PREPARE`
- `DROP STATEMENT`

as well as statements to control execution:

- `EXECUTE`
- `DESCRIBE`
- `DECLARE CURSOR`
- `OPEN (cursor)`
- `FETCH (cursor)`
- `PUT (cursor)`
- `CLOSE (cursor)`

The support program may use `CREATE PROGRAM` and Extended `PREPARE` to build an access module and preprocess the end-user SQL statements. However, you must first `PREPARE` the support program itself. This is done by running it

through the SQL/DS Assembler Preprocessor and the Assembler. Refer to Figure 30.



H3

Figure 30. An Example of an Interpretive Support Program for Building and Executing SQL Statements in an Access Module

The resulting support program can accept end-user SQL statements and can create access modules in the SQL/DS data base to hold them. There can be a separate access module to hold the SQL statements for each end-user, for example. A more advanced support program may even accept end-user commands that are a higher level than that supported by SQL/DS, then translate them to SQL statements before preparing them.

Notice the distinction between access modules SP and P. SP is created by the SQL/DS Assembler preprocessor for the SQL statements of the support program, while P is built by the support program (via CREATE PROGRAM) for the particular end-user SQL statements.

If the support program allows both adding SQL statements to, and dropping SQL statements from the access module P, then the support program must utilize and be preprocessed with a DROP STATEMENT, as well as the PREPARE. Of course there are a few other ordinary SQL statements that may be appropriate for the support program: WHENEVER, COMMIT/ROLLBACK WORK, and so on to make it complete.

So far this example has not addressed execution of the end-user SQL statements. We have already listed the extended dynamic statements that support execution (Extended EXECUTE, DECLARE CURSOR, and so on). The support program would ordinarily support end-user commands to retrieve data and update data (again either direct SQL/DS statements or higher level commands that require translation to SQL/DS). The addition of this execution capability does not alter

the concept shown in Figure 30 except to add additional extended dynamic statements to the support program.

The DESCRIBE statement can be used in basically the same way as illustrated under normal dynamic statements (“Dynamically Defined Statements” on page 147).

Note that only one “copy” of each extended dynamic statement need be provided in the support program, because each of these statements is parameterized with host variables that can be dynamically changed for each use. For example, one DECLARE CURSOR statement may service all cursor retrievals, even if they are concurrently open, since each can be given a different cursor name via the host variable value for the cursor name. The support program must translate user requests into SQL statements, including the initialization of host variables required.

Structurally this example is fairly straight forward. We have assumed that the support program remains in control as an interpreter through preparation, maintenance, and execution of the user’s SQL statements.

For a typical language preprocessor program such as those provided with SQL/DS, this is not the case. If you write a support program for a new language preprocessor, you would probably separate the two parts, each with SQL statements:

1. One for preparation of end-user SQL statements and creation of an access module
2. Another for supporting the execution of the SQL statements that were prepared by the first part.

The SQL functions performed are about the same as in the previous example, except we will assume that no access module maintenance functions are needed. The language preprocessor has the following characteristics:

1. It is a batch program, rather than an interpreter.
2. Since it requires extended dynamic statements, it is written in Assembler. This was also true in the preceding example. Alternatively, at least part of it must be in written in Assembler (the part that contains the extended dynamic statements) and the remaining part must be written in a language that is capable of calling an Assembler module.
3. Rather than accepting predefined commands from the end-user, the end-user’s source language code is scanned for SQL statements, which must be identified by some defined convention (EXEC SQL, for example) for proper recognition.
4. The support program must record information about host program variables in a control structure that is added to the end-user’s source program and passed via a generated call to the execution-time part of the support program. This control structure is used to build SQLDA structures that are passed to or received from SQL/DS (refer to the Extended EXECUTE, OPEN, and FETCH statements).

5. The execution part of the support program is link/loaded with the user's application program, where it is available to handle the execution-time functions.
6. As each end-user SQL statement is prepared, the program-id and the statement-id (returned by SQL/DS along with the program-id) must be saved in a control structure (again generated into the end-user's source program) for use by the execution-time support program.
7. For each SQL statement in the end-user's source program, a call must be generated to the execution-time support program, passing the control structure, containing the host variable, program-id, and statement-id information for the current SQL statement.
8. The execution-time support program must build the SQLDA structures required, set values in host variables required by the execution-time extended dynamic statements, and then execute these statements.

This process is illustrated in Figure 31, Figure 32, Figure 33, and Figure 34. The support program is the preprocessor for language X. It preprocesses the end-user program, modifying the source (adding control structures and generating calls to pass to the support program part two at execution-time). Once the modified end-user source has been compiled by the Language X compiler, it is combined in one load module with the object code for the support program part two, which provides the SQL/DS support for execution-time functions (DECLARE CURSOR, and so on).

Figure 31 shows the SQL/DS preprocessing and assembly steps for the two parts of the support program. This results in two access modules: SP1 and SP2.

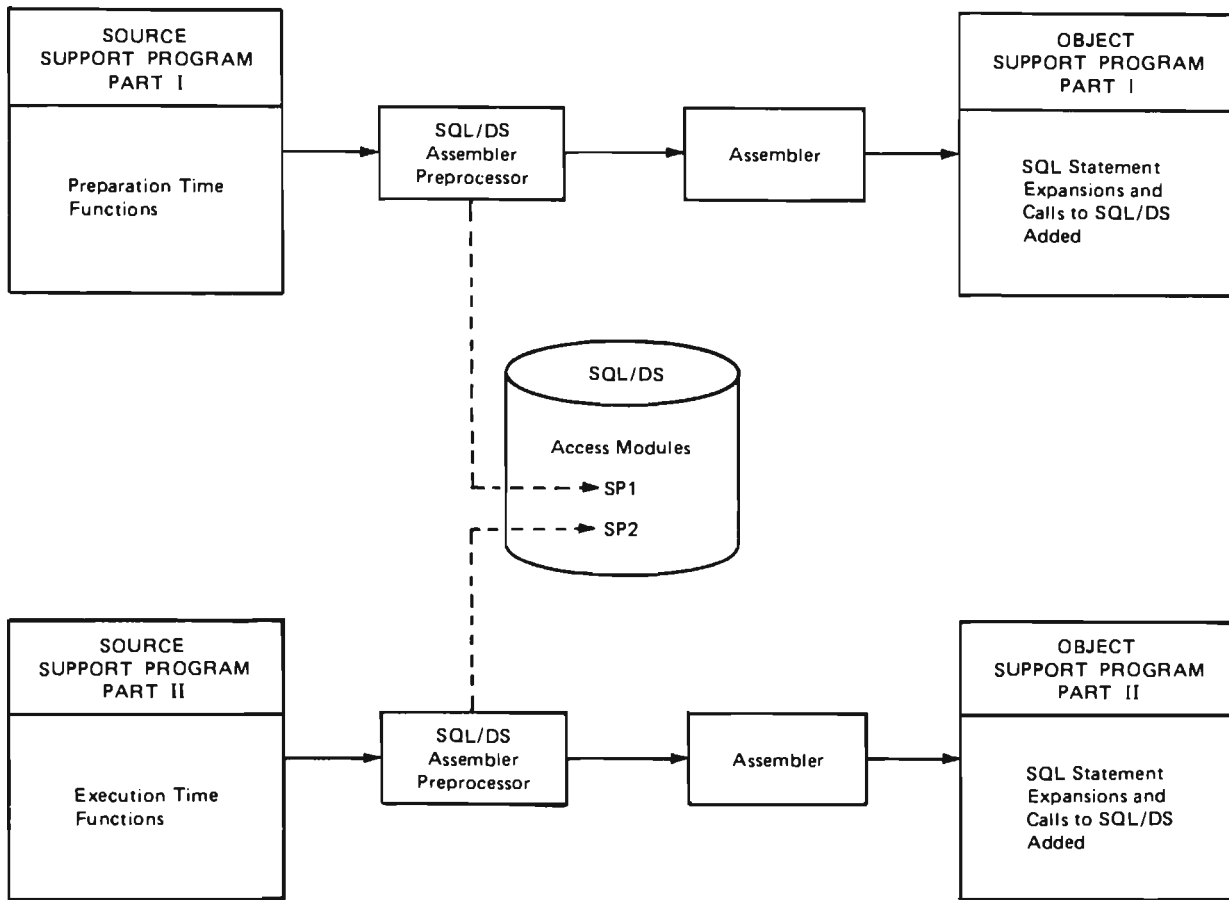


Figure 31. Preprocessing and Assembly of a Two-Part Support Program

Figure 32 shows how the two resulting object modules of the support program are used to process end-user SQL statements.

Preprocessing End-User Program, P.

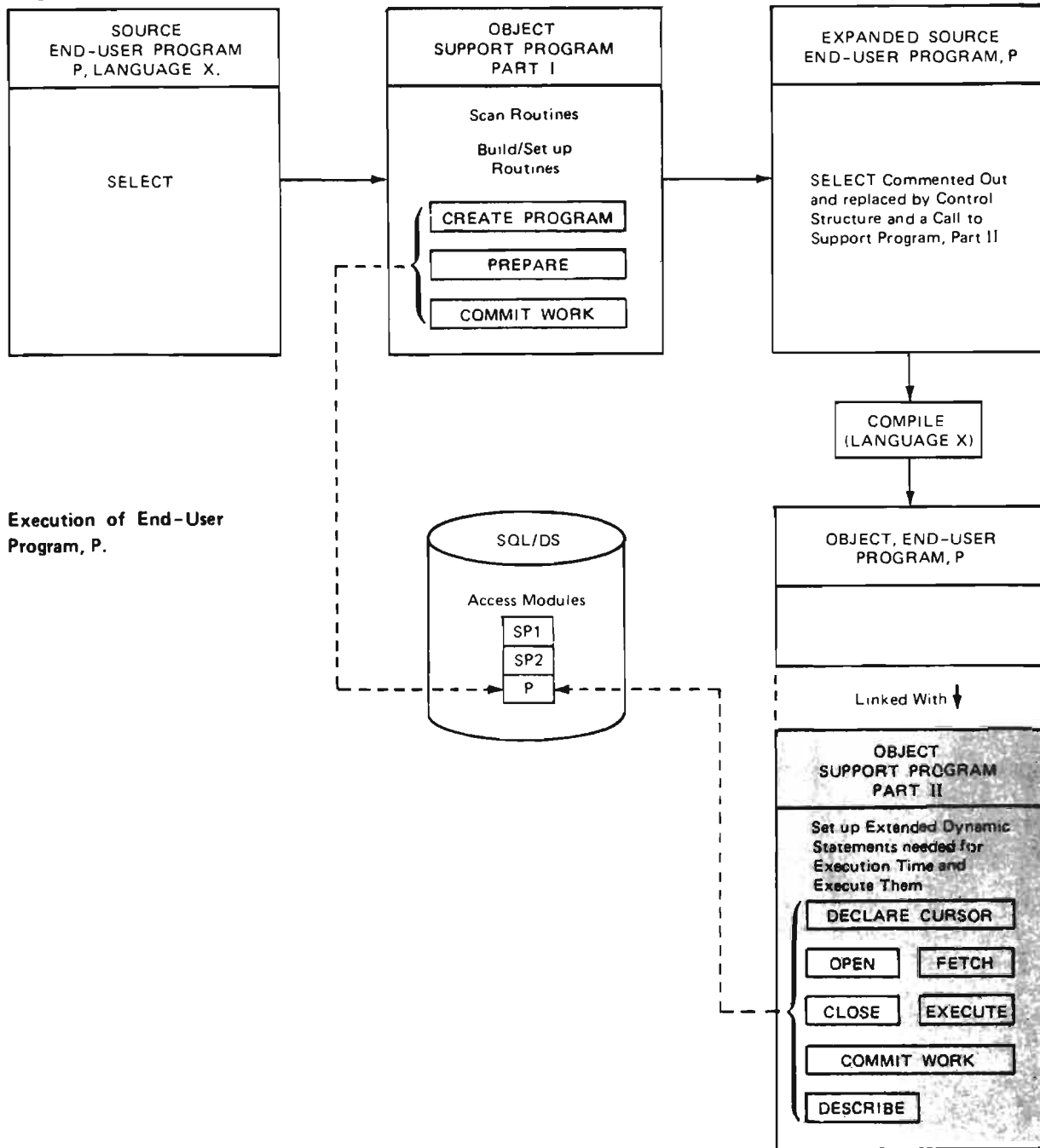


Figure 32. Preprocessing and Execution of an End-User Program by a Two-Part Support Program

Part 1 scans the end-user's source for SQL statements; uses CREATE PROGRAM to build an empty access module, P; uses Extended PREPARE to add SQL statements to P; and uses COMMIT WORK to finalize P. It also adds calls and control structures, required by Part 2 of the support program, to the user's source program and comments out the original SQL statement.

Part 2 of the support program works with the access module, P, executing the SQL statements scanned and prepared by Part 1. Part 2 uses the control structures passed in the calls generated by Part 1.

Part 2 must be link/loaded with any end-user module that is preprocessed by Part 1.

Figure 33 shows Part 1 of the support program in more detail, with pseudo code to illustrate a simple user program that includes a DECLARE ...CURSOR FOR SELECT....., an OPEN of that cursor, and a FETCH for the same cursor. Control structures are shown in more detail and some particular values for parameters are given. The value 26 returned from the PREPARE statement is only for purposes of illustration, representing a unique identifier returned by SQL/DS to identify the statement within the access module, P. A userid may be necessary to identify the owner of the access module, but it is omitted here for simplicity. Other statements, such as CLOSE (cursor) and COMMIT WORK are not shown in order to simplify the illustration.

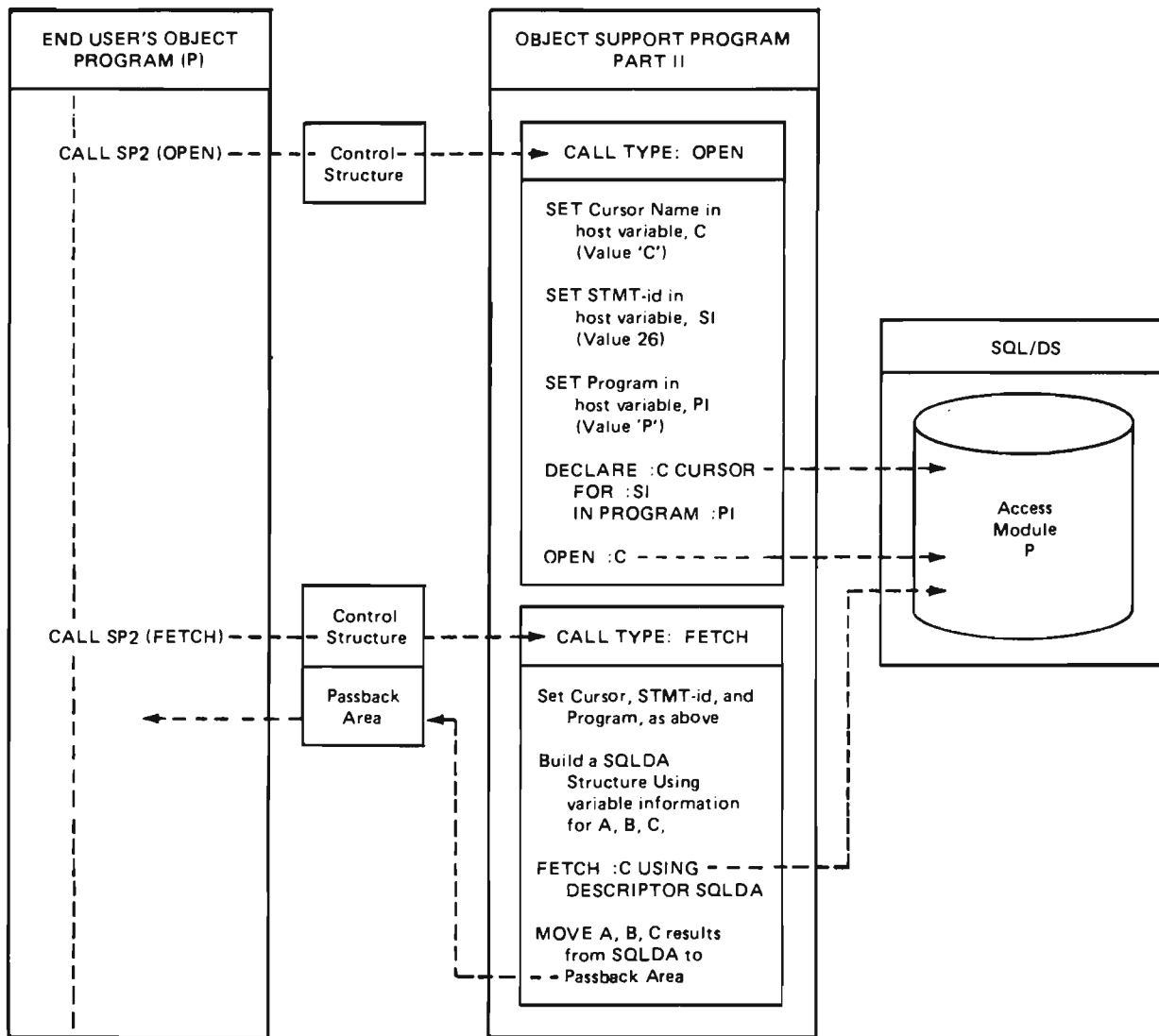


Figure 34. Pseudo-Code Example of Execution of End-User Program P

Logical Unit of Work Considerations

There are primarily three cases to consider when determining the proper grouping of extended dynamic statements in a logical unit of work:

1. A Logical Unit of Work contains a CREATE PROGRAM without the MODIFY Option. This would be the case for a language preprocessor application.
2. A logical unit of work contains a CREATE PROGRAM with the MODIFY Option. This would be the case for an application that gets new SQL statements from its users, then prepares and executes them immediately (but

also has them available for later execution, since they are stored in an access module).

3. A logical unit of work contains no **CREATE PROGRAM** (the referenced access module has been created with the **MODIFY** option in another logical unit of work). This would be the case for an application that prepares, executes or changes statements in an access module that was created previously.

In the first case, the only other extended dynamic statement permitted is the **PREPARE** statement and it must reference only the program that is specified in the **CREATE PROGRAM** statement. If the logical unit of work is terminated by a **COMMIT WORK**, an SQL/DS access module is created. If no Extended **PREPARE**s were executed, the access module is essentially empty and the **COMMIT WORK** results in an error (-759 SQLCODE). If a **ROLLBACK WORK** statement terminates the logical unit of work, no access module is created. In Figure 35, example 1 is a valid illustration of this case.

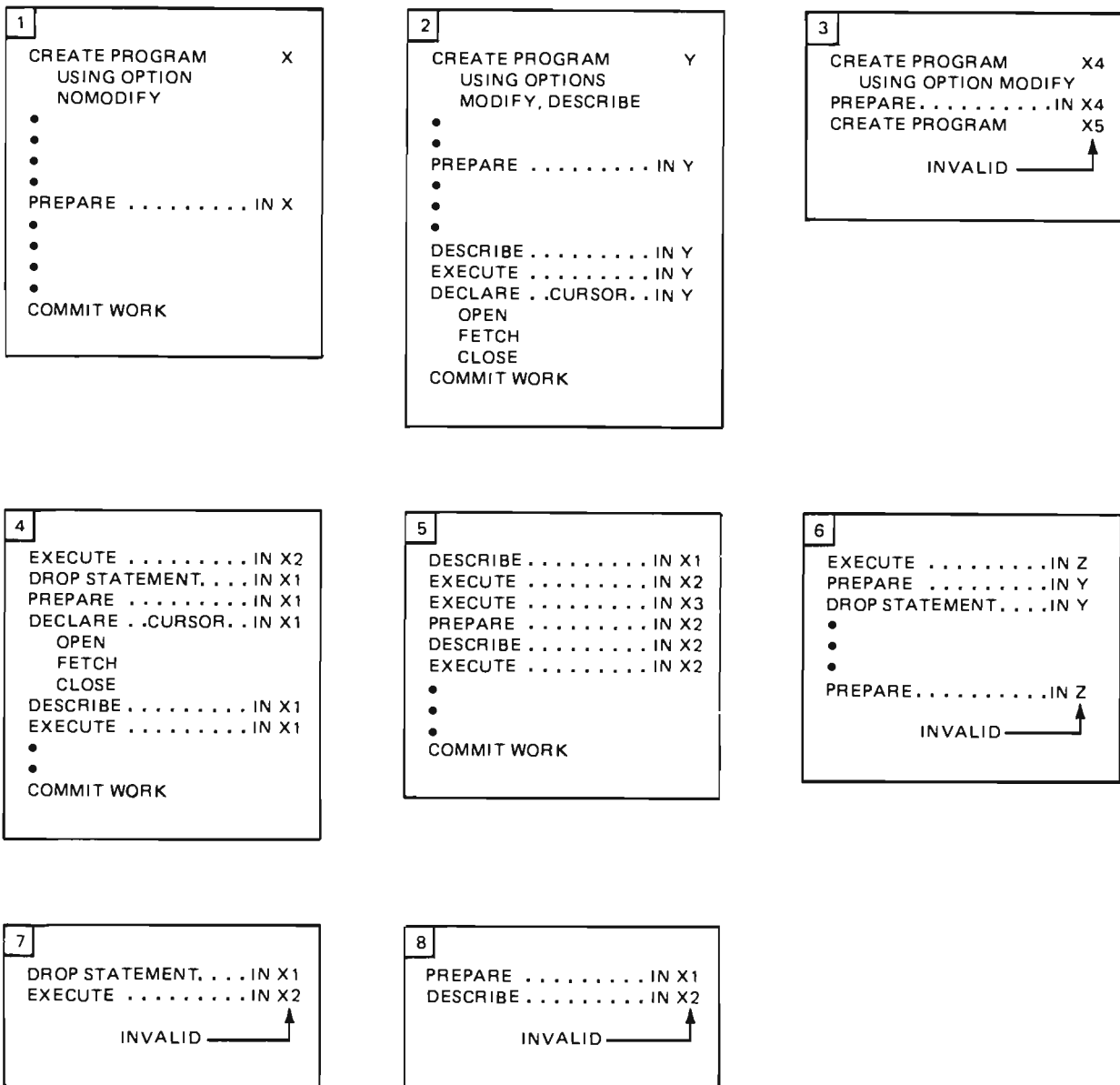


Figure 35. Placement of Extended Dynamic Statements in Logical Units of Work

In the second case, the rules discussed above for case 1 apply, but additionally, Extended DESCRIBE, EXECUTE, DECLARE CURSOR, OPEN, FETCH, DROP STATEMENT, and CLOSE statements may be used in the same logical unit of work, referencing the statements just added to or already contained in the current access module. However, you cannot reference an access module other than the one created in the current logical unit of work. In Figure 35, example 2 is a valid example of this case. Example 3 illustrates an invalid case 2 sequence. If the current logical unit of work is committed before Extended PREPAREs are used to add statements to it (it is empty), it still may be extended in a later logical unit of work (since it is modifiable, it may make sense to leave it empty initially).

In case 3, where the current logical unit of work contains no CREATE PROGRAM, extended dynamic statements may reference any access module that

has been created with a `CREATE PROGRAM` statement. However, once an extended dynamic statement that causes modification of the access module is used (`Extended PREPARE` or `DROP STATEMENT`), then subsequent extended dynamic statements in the same Logical Unit of Work may only refer to the modified access module. Once the logical unit of work is terminated, reference to any access module that has been created by a `CREATE PROGRAM` may be resumed. (Note that this does not preclude additional restrictions: to modify an access module, you must have created it with the `MODIFY` option and to `DESCRIBE` a statement in an access module, it must have been created with the `DESCRIBE` option).

For example, if access modules X1, X2, and X3 have been created with a `CREATE PROGRAM`, where X1 and X2 have the `MODIFY` and `DESCRIBE` options, examples 1, 2, 4 and 5 in Figure 35 are valid while 3, 6, 7, and 8 are invalid.

Virtual Storage Considerations in a Logical Unit of Work

If virtual storage consumption is an important consideration, you should be especially careful when adding statements to a modifiable access module (created with the `MODIFY` option). Approximately 23K is added for each `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement that is added in a logical unit of work. For other statement types only about 16K is added. Since this storage consumption is accumulated until the logical unit of work is terminated by a `COMMIT` or `ROLLBACK WORK`, you may have to be careful to terminate logical units of work more frequently when your program involves this type of maintenance activity. This virtual storage consumption occurs so that you may optionally execute the statements that you just added without first storing the access module in the data base. If you `COMMIT WORK` before executing the new statement, virtual storage consumption is considerably less, but there is additional processing to store the updated access module in the data base and retrieve it again. This trade-off must be made based on the nature of the processing in your program.

Extended Dynamic Statement Descriptions

CREATE PROGRAM

Format:

```
CREATE PROGRAM [userid.] program-name [USING {OPTION | OPTIONS}
                                         {{KEEP | REVOKE}
                                         {DESCRIBE | NODESCRIBE}
                                         {REPLACE | NEW}
                                         {MODIFY | NOMODIFY}
                                         {BLOCK | NOBLOCK}}]
```

Example:

```
CREATE PROGRAM JERRY.MUSICIANS USING OPTIONS DESCRIBE NEW BLOCK
```

Authorization:

Anyone connected to SQL/DS is authorized to issue this statement.

This statement creates an access module. The access module is stored in the data base when a COMMIT WORK is issued.

If the *program-name* is identical to the name of an existing access module (REPLACE option), the existing access module is implicitly dropped and replaced with a new access module.

userid

is an optional parameter that determines the owner of the program that is being created. Userid can be specified as either a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant. If userid is not specified, the userid of the user explicitly or implicitly connected with SQL/DS at run-time is used.

program-name

is the name of the access module that is to be created. Program-name can be specified as a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant.

KEEP | REVOKE

This parameter applies if the access module has previously been created and the creator of the access module has granted the RUN privilege on the resulting access module to other users. KEEP causes these grants of the RUN privilege to remain in effect when the new access module is created. If the REVOKE parameter is specified, or if the creator of the access module is not entitled to grant all privileges embodied in the program, the preprocessor

revokes all existing grants of the RUN privilege. The KEEP and REVOKE parameters are optional; KEEP is the default.

DESCRIBE | *NODESCRIBE*

This parameter allows the use of the Extended DESCRIBE for statements added to the created access module. If DESCRIBE is specified, Extended DESCRIBE statements can be executed for prepared SELECT statements in the access module; if NODESCRIBE is specified, these statements cannot be executed. The DESCRIBE and NODESCRIBE parameters are optional; NODESCRIBE is the default.

REPLACE | *NEW*

This parameter specifies whether the access module being created is new or whether it will replace an existing access module that has the same name. If NEW is specified, an error (SQLCODE -168) results if an access module already exists with the same name. If REPLACE is specified and no previous module exists with the same name, no error or warning is given. If NEW is specified along with KEEP or REVOKE, an error results (SQLCODE -160). The REPLACE and NEW parameters are optional; REPLACE is the default.

MODIFY | *NOMODIFY*

This parameter specifies whether the created access module can be modified once it is stored through a COMMIT WORK. Statements are added to the access module by using the Extended PREPARE and deleted by using the DROP STATEMENT function.

Statements in access modules created with the MODIFY option can also be executed or dropped before committing the logical unit of work in which they were prepared.

The MODIFY parameter should not be used if the entire access module will be replaced using the REPLACE option. Once an access module has been created with the MODIFY parameter specified, it can be changed but not replaced by subsequent CREATE PROGRAM statements. To replace an access module created with the MODIFY option, it is necessary to issue a DROP PROGRAM statement and then issue a CREATE PROGRAM. MODIFY and NOMODIFY are optional parameters; NOMODIFY is the default.

BLOCK | *NOBLOCK*

When the BLOCK parameter is specified, SQL/DS inserts and retrieves rows in groups. This improves performance for programs running in multiple user mode where many rows will be inserted or retrieved. NOBLOCK tells SQL/DS not to group rows. The default is NOBLOCK. BLOCK and NOBLOCK are optional parameters. See “To Block or Not to Block?” on page 254 for guidelines on deciding which programs to specify blocking for.

When the logical unit of work, in which the CREATE PROGRAM statement is issued, is committed (via COMMIT WORK), a new access module is created. ROLLBACK WORK prevents the storage of the new access module. An access module created with the MODIFY option can be committed even if it contains no statements.

Extended PREPARE

Format:

```
PREPARE FROM string-spec SETTING statement-id  
IN [userid.] program-name [USING DESCRIPTOR structure-spec]
```

Example:

```
PREPARE FROM :XSTRING SETTING :STMID  
IN :USERID.:PROGNAME USING DESCRIPTOR MYSQLDA
```

Authorization:

The user executing the PREPARE command must be the creator of the access module (specified program-name) or have DBA authority. Authority for commands prepared are as specified for individual commands in this publication.

This statement permits a statement to be prepared and stored in an access module for later execution. The various extended dynamic statements associated with the PREPARE statement can be issued with program names and statement identifiers. This permits the user to issue these extended dynamic statements independently to serve different programs at different times.

The PREPARE statement is used to add an SQL statement to an existing access module. If the access module is new, the PREPARE statement must be preceded by a CREATE PROGRAM statement. Existing access modules, created using the MODIFY option of CREATE PROGRAM, can be extended using this PREPARE statement.

string-spec

specifies the statement that is to be prepared. String-spec must be a host variable that is a varying string of maximum length 8192.

statement-id

is a host program variable of the INTEGER data type. It is set by SQL/DS to an identifier for the statement that is prepared. It is used in subsequent DESCRIBE, EXECUTE, DROP STATEMENT, and DECLARE CURSOR statements to specify the corresponding prepared statement.

userid

is an optional parameter that identifies the owner of the program that is being modified or extended. Userid can be specified as a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant. If userid is not specified, the user connected to SQL/DS when the PREPARE command is executed becomes the default user. The explicit or default userid is necessary to identify completely the access module being changed.

program-name

is the name of the access module in which the prepared statement is to be stored. If the access module does not exist (has not been created), an error (SQLCODE -805) will result. Program-name can either be a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant.

structure-spec

identifies an input SQLDA structure. When used with the Extended PREPARE, only the following fields are used: SQLD, SQLTYP, and SQLLEN.

Use of SQLDA is optional. It should be used to improve run-time performance and reduce conversions in those cases where data types and lengths are known for the '?' parameters in the prepared SQL statement. The fields described in the SQLDA should match the '?' parameters in the statement being prepared. If there are fewer fields specified in the SQLDA, an error will result; if there are more, excess SQLDA fields will be ignored by SQL/DS. If the type and length information given in the PREPARE SQLDA does not match that given in the EXECUTE or OPEN statements, errors may result.

The following SQL statements cannot be prepared, that is, may not be used for string-spec:

WHENEVER	EXECUTE
INCLUDE	EXECUTE IMMEDIATE
COMMIT WORK	OPEN
ROLLBACK WORK	FETCH/PUT
BEGIN DECLARE SECTION	CLOSE
END DECLARE SECTION	DROP STATEMENT
CREATE PROGRAM	CONNECT
PREPARE	DECLARE CURSOR
DESCRIBE	SELECT with an INTO clause

A question mark (?) can appear in an SQL statement to be "prepared" in any place that a host variable can appear, with some exceptions. These exceptions are discussed under "PREPARE" on page 172.

Because a DBA can add a statement to an access module on behalf of the owner (creator) of the module, where the owner is not authorized for the added function, the DBA should grant the proper authorization to the owner.

Extended DESCRIBE

<p><i>Format:</i></p> <pre>DESCRIBE statement-id IN [userid.]program-name INTO structure-spec [USING {NAMES LABELS BOTH ANY}]</pre>
<p><i>Example:</i></p> <pre>DESCRIBE :STMID IN :USERID.:PROGNAME INTO MYSQLDA</pre>
<p><i>Authorization:</i></p> <p>At execution time, the connected user must be the creator of the access module (program-name), or have DBA authority, or have RUN authority on the access module.</p>

This statement permits the retrieval of information about an SQL SELECT statement prepared by an Extended PREPARE statement. The DESCRIBE command does not have to be in the same logical unit of work or program as the PREPARE statement that was originally used to process the statement. To be successful, the access module must have been created using a CREATE PROGRAM with the DESCRIBE option. DESCRIBE returns the number of fields in the answer set, the data types, lengths, and names in the named SQLDA structure.

statement-id

is specified by the user to contain the statement-id returned by SQL/DS as the result of an earlier PREPARE statement. It indicates the statement that is to be the subject of this DESCRIBE function. Statement-id must be a host program variable of data type INTEGER.

userid

is an optional parameter that identifies the owner of the program in which the referenced statement resides. Userid can be specified as a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant. If userid is not specified, the user connected to SQL/DS when the DESCRIBE statement is executed becomes the default user.

program-name

is the name of the access module in which the referenced SQL statement resides. If the qualified program-name does not refer to an existing access module, an error (SQLCODE -805) will result. Program-name can either be a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant.

structure-spec

identifies an output SQLDA structure that is to receive information about the columns that are to be retrieved by the described SQL statement. When

used with the Extended DECLARE, only the following fields are used: SQLD, SQLTYP, and SQLLEN. Structure-spec is identical to that used for the DESCRIBE dynamic statement (see “Dynamically Defined Statements” in Chapter 5).

The USING clause of the EXTENDED DESCRIBE statement works the same as in the DESCRIBE dynamic statement, and follows the same rules. (See “DESCRIBE” on page 177 for more information.) The labels returned in the SQLDA are those which were in the SYSCOLUMNS catalog table when the SQL statement was prepared.

Extended EXECUTE

Format:

```
EXECUTE statement-id IN [userid.]program-name  
  [USING DESCRIPTOR structure-spec]
```

Example:

```
EXECUTE :STMID IN :USERID.:PROGNAME  
  USING DESCRIPTOR MYSQLDA
```

Authorization:

At execution time, the connected user must be the creator of the access module (program-name), or have DBA authority, or have RUN authority on the access module.

This statement causes SQL/DS to execute a statement (identified by statement-id) that was prepared previously using an Extended PREPARE statement. When the statement is executed, the host variables you specify in your SQLDA are substituted, in order, into the the statement in place of the '?' parameters that were given in the PREPARE statement. Each variable must be of a data type that is compatible with its usage in the "prepared" SQL statement. Extended EXECUTE will fail if the prepared statement was a SELECT statement (requires a DECLARE CURSOR).

statement-id

is specified by the user to contain the statement-id returned by SQL/DS as the result of an earlier PREPARE statement. It indicates the statement that is to be executed. Statement-id must be a host program variable of data type INTEGER.

userid

is an optional parameter that identifies the owner of the program in which the referenced statement resides. Userid can be specified as a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant. If userid is not specified, the user connected to SQL/DS when the EXECUTE statement is executed becomes the default user.

program-name

is the name of the access module in which the referenced SQL statement resides. If the qualified program-name does not refer to an existing access module, an error (SQLCODE -805) will result. Program-name can either be a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant.

structure-spec

identifies an input SQLDA structure that provides information about variables that were not specified when the statement was prepared ('? parameters). For additional information on SQLDA and its use, see "Dynamically Defined Statements" on page 147.

Extended DECLARE CURSOR

Format:

```
DECLARE cursor-name CURSOR FOR statement-id IN [userid.]program-name
```

Example:

```
DECLARE :CURSOR1 CURSOR FOR :STMID IN :USERID.:PROGNAME
```

Authorization:

At execution time, the connected user must be the creator of the access module (program-name), or have DBA authority, or have RUN authority on the access module.

This statement declares the cursor through which a user may OPEN, FETCH, PUT, or CLOSE the results of a prepared statement. Cursors are associated with a prepared SELECT statement or INSERT statement by the statement-id and the program-name specified in the DECLARE CURSOR statement. Extended DECLARE CURSOR may be used for any SELECT or INSERT statement in an access module created using CREATE PROGRAM. A cursor need not be declared in the same logical unit of work or program in which the statement was prepared.

Remember that a cursor name used in a WHERE CURRENT OF clause cannot be specified via a host variable. Therefore, you should make sure that, at execution time, the content of cursor-name in the EXTENDED DECLARE CURSOR statement must be the same as the cursor-name hard-coded in the WHERE CURRENT OF clause.

cursor-name

identifies the cursor(s) that are to be used. Cursor-name is a host variable, thus allowing a cursor name to be provided when the program is run. Cursor-name must be a VARCHAR data type. The name placed in the cursor-name host variable must be unique within the logical unit of work in which it is used, because that is the scope of the name.

statement-id

is specified by the user to contain the statement-id returned by SQL/DS as the result of an earlier PREPARE statement. It indicates the statement that is to be executed. Statement-id must be a host program variable of data type INTEGER.

userid

is an optional parameter that identifies the owner of the program in which the referenced statement resides. Userid can be specified as a host variable (fixed-length, eight characters, padded to the right with blanks) or a

constant. If `userid` is not specified, the user connected to SQL/DS when the `DECLARE` statement is executed becomes the default user.

program-name

is the name of the access module in which the referenced SQL statement resides. If the qualified `program-name` does not refer to an existing access module, an error (SQLCODE -805) will result. `Program-name` can either be a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant.

Once the `DECLARE CURSOR` statement is issued, a cursor is established; the cursor can then be opened and used to retrieve or insert rows through the `OPEN`, `FETCH`, and `PUT` statements.

Extended OPEN

Format:

```
OPEN cursor-name [USING DESCRIPTOR structure-spec]
```

Example:

```
OPEN :CURSOR1 USING DESCRIPTOR MYSQLDA
```

Authorization:

At execution time, the connected user must be the creator of the access module (program-name), have DBA authority, or have RUN authority on the access module.

This statement opens the cursor of a previously prepared statement to retrieve the results of a query, or to insert values into SQL/DS.

In many respects, the Extended OPEN statement is similar to the OPEN statement for dynamically defined queries, described in Chapter 2. However, in the Extended OPEN statement, the cursor-name is a host variable, thereby making it possible for a user to provide the cursor name when the program is run and to open the cursor in a logical unit of work or program other than the one in which the statement was prepared. DECLARE CURSOR and OPEN must occur in the same logical unit of work.

Note: When the cursor you are opening is to be used for inserting data into a table, the USING DESCRIPTOR clause should not be included.

cursor-name

identifies the cursor to be opened. Cursor-name is a host variable, thus allowing a cursor name to be provided when the program is run.

Cursor-name must be a VARCHAR data type. The name placed in the cursor-name host variable must be unique within the logical unit of work in which it is used, because that is the scope of the name.

structure-spec

identifies an SQLDA structure that provides information concerning input variables that were specified as '?' parameters when the statement was prepared. For additional information on SQLDA and its use, see "Dynamically Defined Statements" on page 147.

Extended FETCH

Format:

```
FETCH cursor-name USING DESCRIPTOR structure-spec
```

Example:

```
FETCH :CURSOR1 USING DESCRIPTOR MYSQLDA
```

Authorization:

At execution time, the connected user must be the creator of the access module (program-name), have DBA authority, or have RUN authority on the access module.

This statement retrieves one row of a query result defined by a PREPARE statement. The indicated cursor must be declared and opened. The places into which the individual fields are to be fetched are indicated by the structure-spec.

In most respects, the Extended FETCH statement is identical to the FETCH statement for dynamically defined queries, as described under “Dynamically Defined Statements” on page 147. However, in the Extended FETCH statement, the cursor-name is a host variable, thereby making it possible for a user to provide the cursor name when the program is run and to FETCH in a logical unit of work or program other than the one in which the statement was prepared. DECLARE CURSOR, OPEN, and FETCH must occur in the same logical unit of work.

cursor-name

identifies the cursor that is used. Cursor-name is a host variable, thus allowing a cursor name to be provided when the program is run.

Cursor-name must also have a data type of VARCHAR. The name placed in the cursor-name host variable must be unique within the logical unit of work in which it is used, because that is the scope of the name.

structure-spec

identifies an SQLDA structure that provides information concerning output variables that were specified as ‘?’ parameters when the statement was prepared. For additional information on SQLDA and its use, see “Dynamically Defined Statements” on page 147.

Extended PUT

Format 1:

```
PUT cursor-name FROM host-variable-list
```

Format 2:

```
PUT cursor-name USING DESCRIPTOR structure-spec
```

Examples:

```
PUT :CURSOR1 FROM :X, :Y  
PUT :CURSOR2 USING DESCRIPTOR SQLDA
```

Authorization:

At execution time, the connected user must be the creator of the access module (program-name), have DBA authority, or have RUN authority on the access module.

This statement inserts one row of data defined by a PREPARE statement. The indicated cursor must be declared and opened. The sources of the individual fields to be inserted are indicated by the structure-spec or the host-variable-list.

In most respects, the Extended PUT statement is identical to the PUT statement described under "PUT Statement for Dynamically Defined Inserts" on page 181. However, in the Extended PUT statement, the cursor-name is a host variable. This feature makes it possible for a user to provide the cursor name when the program is run and to issue a PUT statement in a logical unit of work or program other than the one in which the statement was prepared. DECLARE CURSOR, OPEN, and PUT must occur in the same logical unit of work.

cursor-name

identifies the cursor that is used. Cursor-name is a host variable, thus allowing a cursor name to be provided when the program is run. Cursor-name must also have a data type of VARCHAR. The name placed in the cursor-name host variable must be unique within the logical unit of work in which it is used, because that is the scope of the name.

host-variable-list

identifies the values to be inserted. The host variables provide information concerning input variables that were specified as '?' parameters when the statement was prepared.

structure-spec

identifies an SQLDA structure that provides information concerning input variables that were specified as '?' parameters when the statement was prepared. For additional information on SQLDA and its use, see "Dynamically Defined Statements" on page 147.

Extended CLOSE

Format:

```
CLOSE cursor-name
```

Example:

```
CLOSE :CURSOR1
```

Authorization:

At execution time, the connected user must be the creator of the access module (program-name), have DBA authority, or have RUN authority on the access module.

This statement “closes” the cursor identified by cursor-name. The cursor leaves the open state, and its active set becomes undefined. No FETCH or PUT statement can be executed on the cursor, and no DELETE or UPDATE statement can refer to its current position, until the cursor is reopened by an OPEN statement. CLOSE permits SQL/DS to release the resources associated with maintaining an open cursor.

In most respects, the Extended CLOSE statement is identical to the CLOSE statement described under “Retrieving or Inserting Data with a Cursor” on page 19. However, in the Extended CLOSE statement, the cursor-name is a host variable, thereby making it possible for a user to provide the cursor name when the program is run and to CLOSE the cursor in a logical unit of work or program other than the one in which the statement was prepared. CLOSE must occur in the same logical unit of work as the corresponding cursor declaration.

cursor-name

identifies the cursor that is to be closed. Cursor-name is a host variable, thus allowing a cursor name to be provided when the program is run.

Cursor-name must be a VARCHAR data type. The name placed in the cursor-name host variable must be unique within the logical unit of work in which it is used, because that is the scope of the name.

DROP STATEMENT

Format:

```
DROP STATEMENT statement-id IN [userid.]program-name
```

Example:

```
DROP STATEMENT :STMID IN :USERID.:PROGNAME
```

Authorization:

At execution time, the connected user must be the creator of the access module (program-name), have DBA authority, or have RUN authority on the access module.

This statement selectively deletes a statement from an access module. When used with the Extended PREPARE statement, it may be used to effectively replace a statement in an access module. (That is, the DROP STATEMENT deletes the statement and the PREPARE statement adds a new statement.) DROP STATEMENT applies only to access modules created with a CREATE PROGRAM statement with the MODIFY option.

statement-id

is specified by the user to contain the statement-id returned by SQL/DS as the result of an earlier PREPARE statement. It indicates the statement that is to be dropped in the named access module. Statement-id must be a host program variable of data type INTEGER.

userid

is an optional parameter that identifies the owner of the program in which the referenced statement resides. Userid can be specified as a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant. If userid is not specified, the user connected to SQL/DS when the DROP statement is executed becomes the default user.

program-name

is the name of the access module in which the referenced SQL statement resides. If the qualified program-name does not refer to an existing access module, an error (SQLCODE -805) will result. Program-name can either be a host variable (fixed-length, eight characters, padded to the right with blanks) or a constant.

Appendix A. SQL/DS Reserved Words

The following keywords are reserved in the SQL language. You cannot use them in SQL statements except for:

1. Their defined meaning in the SQL syntax; and,
2. As host variables (preceded by a colon).

In particular, you cannot use them as names for tables, indexes, columns, views, or DBSPACES unless they are enclosed in double quotes ("") as described in Chapter 1.

ACQUIRE	EXCLUSIVE	NAMED
ADD	EXISTS	NHEADER
ALL	EXPLAIN	NOT
ALTER		NULL
AND	FLOAT	OF
ANY	FOR	ON
AS	FROM	OPTION
ASC		OR
AVG	GRANT	ORDER
	GRAPHIC	
BETWEEN	GROUP	PAGE
BY		PAGES
	HAVING	PCTFREE
CHAR		PCTINDEX
COLUMN	IDENTIFIED	PRIVATE
COMMENT	IN	PRIVILEGES
COMMIT	INDEX	PROGRAM
CONNECT	INSERT	PUBLIC
COUNT	INTEGER	
CREATE	INTO	RELEASE
CURRENT	IS	RESOURCE
		REVOKE
DBA	LIKE	ROLLBACK
DBSPACE	LOCK	ROW
DECIMAL	LONG	RUN
DELETE		
DESC	MAX	SCHEDULE
DISTINCT	MIN	SELECT
DROP	MODE	SET
		SHARE

SMALLINT
STATISTICS
STORPOOL
SUM
SYNONYM

TABLE
TO

UNION
UNIQUE
UPDATE
USER

VALUES
VARCHAR

VARGRAPHIC
VIEW

WHERE
WITH
WORK

- **Maximum length of a host variable is 18 characters, unless the host language has a further restriction. For example, FORTRAN permits only six-character host variable names.**
- **Maximum number of application program variables used in SQL statements in one preprocessed program is 512.**
- **Maximum number of host variables used in one SQL statement is 256.**
- **Maximum length of one SQL statement is 8192 characters.**
- **Maximum number of columns in an index definition is 16.**
- **When an index is created, the sum of the lengths of the indexed columns, plus 25% of the lengths of any indexed columns that are of varying-length character type, must not exceed 255 bytes.**
- **Maximum number of columns listed in an ORDER BY clause is 16.**
- **Maximum number of items that may appear in a select list is 255.**
- **Maximum number of tables in a DBSPACE is 255.**
- **Maximum number of concurrent users is 252.**
- **Maximum number of cursors in a program is 512.**
- **Maximum number of IUCV connections for a virtual machine (MAXCONN) is 65,535; the default is 4.**

Appendix C. PL/I Considerations

This appendix contains a sample program that illustrates the use of SQL within PL/I. Following the sample are specific rules for using SQL in PL/I.

ARISPLIC -- PL/I Sample Program

ARISPLIC is a PL/I sample program that is provided with SQL/DS for VM/SP systems. The source code of this program begins on the next page. You can learn most of the rules for using SQL within PL/I just by scanning through the program. Note, in particular, how the program satisfies the requirements of the application prolog and epilog. Near the beginning of the program all the host variables are declared, error handling is defined, and a connection is established with SQL/DS. Near the logical end of the program, the data base changes are committed. (The connection to SQL/DS is implicitly released on program termination.)

Notice that the delimiters for all SQL statements coded within a PL/I program are "EXEC SQL" and a semicolon (for example, "EXEC SQL INCLUDE SQLCA;").

The DCL statements for the host variables were determined by referring to Figure 40 on page 383. That figure gives the PL/I representation for each of the ten SQL/DS data types. When you are coding your own applications you'll need to obtain the data types of the columns that your host variables interact with. This can be done either by consulting the person who created the table, or by querying the SQL/DS catalogs. The SQL/DS catalogs are described in the *SQL/Data System Planning and Administration for VM/SP* manual.

Notice also that all the host variables are declared in separate DCL statements. You can't declare more than one host variable in a single declare statement. This example is incorrect:

```
EXEC SQL BEGIN DECLARE SECTION;
      DCL USERID CHAR(8), PASS CHAR(8);
      DCL PART BIN FIXED(15), DESC CHAR(24) VAR;
      .
      .
      .
EXEC SQL END DECLARE SECTION;
```

Don't do this! SQL/DS won't recognize PASS and DESC.
--


```

/*****
/*
/* SAMPLE PROGRAM FOR VM/SP      ARISPLIC
/*
/* PURPOSE:          THIS PROGRAM SERVES TWO PURPOSES:
/*                   1. IT IS AN EXAMPLE FOR HOW TO IMBED SQL
/*                   STATEMENTS IN A PL/1 PROGRAM.
/*                   2. IT CAN BE USED TO TEST SOME BASIC SQL
/*                   FUNCTIONS FROM AN APPLICATION PROGRAM.
/*
/* DESCRIPTION:      THIS PROGRAM GENERATES A SAMPLE ORDER FOR
/*                   THE PARTS 302 AND 311 IF QONHAND IS LESS
/*                   THAN 1000 AND 700 RESPECTIVELY, AND IF
/*                   QONORDER IS ZERO.  THE TABLE QUOTATIONS
/*                   IS UPDATED ACCORDINGLY.  PART 302 IS ORDERED
/*                   FROM THE COMPANY THAT SELLS IT FOR THE
/*                   LOWEST PRICE, PART 311 FROM THE COMPANY WITH
/*                   THE SHORTEST DELIVERY TIME.
/*
/*                   AT THE END OF THE PROGRAM PART 321 IS DELETED
/*                   FROM THE DATA BASE.
/*
/* PREREQUISITE:    THE SQL/DS SAMPLE TABLES MUST BE CREATED AND
/*                   LOADED.
/*
/* OUTPUT PRODUCED:  1. AN EXECUTION BEGIN AND END MESSAGE IS
/*                   PRINTED AT THE BEGIN AND END OF PROGRAM
/*                   EXECUTION.
/*                   2. ALL TABLES ARE PRINTED WITH THEIR ORIGI-
/*                   NAL CONTENTS.
/*                   3. A SAMPLE ORDER IS PRINTED.
/*                   4. THE CONTENTS OF THE TABLES ARE PRINTED
/*                   AFTER ALL UPDATES AND DELETES ARE MADE.
/*                   5. UNEXPECTED RETURN CODE:
/*                   AN ERROR MESSAGE IS ISSUED TOGETHER WITH
/*                   THE SQLCA-INFORMATION AND CHANGES BACKED
/*                   OUT.
/*
/*****
SAMPP: PROC OPTIONS (MAIN);
%SKIP;
/*****
/*
/*          ESTABLISH HOST VARIABLES
/*
/*****
%SKIP;
EXEC SQL BEGIN DECLARE SECTION;
%SKIP;
DCL PARTNO DEC FIXED(6);
/* THE DESCRIPTION COLUMN IN THE INVENTORY TABLE IS VARCHAR(24)
/* SQL/DS WILL CONVERT IT TO FIXED LENGTH AND TRUNCATE LONGER
/* DATA WHENEVER THE DESCR HOST VARIABLE IS USED.
DCL DESCR CHAR(10);
DCL QONHAND DEC FIXED(11);
DCL SUPPNO DEC FIXED(6);
DCL NAME CHAR(15);
DCL ADR CHAR(35);
DCL TIME DEC FIXED(6);
DCL QONORDER DEC FIXED(11);
DCL PRICE DEC FIXED(5,2);
DCL ID CHAR(8) INIT('SQLDBA ');
DCL PASSW CHAR(8) INIT('SQLDBAPW');
%SKIP;
EXEC SQL END DECLARE SECTION;
%SKIP;

```

```

/*          INTERNAL PROGRAM VARIABLES          */
%SKIP;
      DCL STMT CHAR(25);
      DCL TPRICE1 DEC FIXED(9,2);
      DCL TPRICE2 DEC FIXED(9,2);

%SKIP;
/*****
/*
/*          OPEN PRINT FILE AND DISPLAY START MSG          */
/*
/*
/*
/*****
%SKIP;
DCL PRTFILE FILE STREAM OUTPUT PRINT;
DISPLAY ('SAMPLE PROGRAM ARISPLIC STARTED');
PUT FILE (PRTFILE) SKIP EDIT ('SAMPLE PROGRAM ARISPLIC STARTED')
      (A(31));

%SKIP;
/*****
/*
/*          ERROR HANDLING          */
/*
/*
/*
/*****
%SKIP;
      EXEC SQL INCLUDE SQLCA;
/* THIS PROGRAM WILL IGNORE WARNING AS THEY WILL NOT AFFECT RESULTS */
      EXEC SQL WHENEVER SQLWARNING CONTINUE;
      EXEC SQL WHENEVER SQLERROR GOTO SQLERR;
      EXEC SQL WHENEVER NOT FOUND GOTO SQLERR;

%SKIP;
/*****
/*
/*          START PROGRAM          */
/*
/*
/*
/*****
%SKIP;
      STMT = 'EXEC SQL CONNECT          ';
      EXEC SQL CONNECT :ID IDENTIFIED BY :PASSW;

%SKIP;
/*****
/*
/*          PRINT TABLES          */
/*
/*
/*
/*****
%SKIP;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE INVENTORY UNCHANGED ***
') (PAGE, A(45));
CALL PRINT1;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE QUOTATIONS UNCHANGED **
*') (PAGE, A(46));
CALL PRINT2;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE SUPPLIERS UNCHANGED ***
') (PAGE, A(45));
CALL PRINT3;
%SKIP;
/*****
/*
/*          FIND MINIMUM PRICE FOR PART #302          */
/*
/*
/*
/* THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM PRICE OF ALL
/* OCCURRENCES OF PART #302 WITH QONHAND LESS THAN 1000, AND
/* QONORDER 0. AS PRICE IS A COLUMN IN QUOTATIONS, AND QONHAND A
/* COLUMN IN INVENTORY, THE TWO TABLES HAVE TO BE LINKED VIA A JOIN
/* BETWEEN THE PART NUMBERS IN INVENTORY AND THOSE IN QUOTATIONS.
/* NO CURSOR IS USED BECAUSE THE STATEMENT IS EXPECTED TO RETURN
/* ONLY ONE ROW.
/*
/*
/*****

```

```

%SKIP;
  STMT = 'SELECT MIN(PRICE) FOR 302';
  EXEC SQL SELECT MIN(PRICE)
    INTO :PRICE
    FROM INVENTORY, QUOTATIONS
    WHERE INVENTORY.PARTNO = 302 AND
          QONHAND < 1000 AND
          INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
          QONORDER = 0;

%SKIP;
/*****
/*
/*      RETRIEVE DATA FOR ORDER AND
/*      UPDATE QONORDER FOR PART #302, TABLE QUOTATIONS
/*
*****/
%SKIP;
  TPRICE1 = 1000 * PRICE;
  STMT = 'SELECT FOR PART #302      ';
  EXEC SQL SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND,
    PRICE, NAME, ADDRESS, QUOTATIONS.SUPPNO
    INTO :PARTNO, :DESCR, :QONHAND, :PRICE, :NAME, :ADR,
        :SUPPNO
    FROM INVENTORY, QUOTATIONS, SUPPLIERS
    WHERE INVENTORY.PARTNO = 302 AND
          INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
          PRICE = :PRICE AND
          QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO;
  STMT = 'UPDATE QUOTATIONS      ';
  EXEC SQL UPDATE QUOTATIONS SET QONORDER = 1000
    WHERE PARTNO = :PARTNO AND
          QONORDER = 0 AND
          PRICE = :PRICE AND
          SUPPNO = :SUPPNO;

%SKIP;
/*****
/*
/*      PRINT SAMPLE ORDER WITH RESULTS OF PART #302
/*
*****/
%SKIP;
  PUT FILE (PRTFILE) EDIT ('SAMPLE ORDER') (PAGE, X(30), A(12));
  PUT FILE (PRTFILE) EDIT ('NUMBER OF', 'DESCRIPTION', 'QUANTITY',
    'COMPANY NAME', 'COMPANY ADDRESS', 'PRICE PER', 'TOTAL')
    (SKIP(2), COL(2), A(9), COL(14), A(11), COL(28), A(8),
    COL(39), A(12), COL(55), A(15), COL(92), A(9), COL(102),
    A(5));

%SKIP;
  PUT FILE (PRTFILE) EDIT ('PARTS', 'ON HAND', 'UNIT', 'COSTS')
    (COL(4), A(5), COL(28), A(7), COL(94), A(4), COL(102), A(5));

%SKIP;
  PUT FILE (PRTFILE) EDIT ('1000', DESCR, QONHAND, NAME, ADR, PRICE,
    TPRICE1) (SKIP(2), COL(3), A(4), COL(14), A(10), COL(24),
    F(11), COL(39), A(15), COL(55), A(35), COL(90), F(9,2),
    COL(100), F(9,2));

%SKIP;
/*****
/*
/*      FIND MINIMUM DELIVERY TIME FOR PART #311
/*
/*      THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM DELIVERY
/*      TIME OF ALL OCCURENCES OF PART #311 WITH QONHAND LESS THAN 700
/*      AND QONORDER 0.  AS DELIVERY TIME IS A COLUMN IN QUOTATIONS
/*      AND QONHAND A COLUMN IN INVENTORY, THE TWO TABLES HAVE TO BE
/*      LINKED VIA A JOIN BETWEEN THE PART NUMBERS IN QUOTATIONS AND
/*      THOSE IN INVENTORY.
*****/

```

```

/*
/*****
%SKIP;
      STMT = 'SELECT MIN(DELIVERY_TIME)';
      EXEC SQL SELECT MIN(DELIVERY_TIME)
            INTO :TIME
            FROM INVENTORY, QUOTATIONS
            WHERE INVENTORY.PARTNO = 311 AND
                  QONHAND < 700 AND
                  INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
                  QONORDER = 0;
%SKIP;
/*****
/*
/*          RETRIEVE DATA OF PART 311 FOR THE ORDER
/*          AND
/*          UPDATE QONORDER FOR PART #311
/*
/*****
%SKIP;
      STMT = 'SELECT FOR PART #311      ';
      EXEC SQL SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND,
            PRICE, NAME, ADDRESS
            INTO :PARTNO, :DESCR, :QONHAND, :PRICE, :NAME, :ADR
            FROM INVENTORY, QUOTATIONS, SUPPLIERS
            WHERE INVENTORY.PARTNO = 311 AND
                  INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
                  DELIVERY_TIME = :TIME AND
                  QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO;
      STMT = 'UPDATE QUOTATIONS      ';
      EXEC SQL UPDATE QUOTATIONS SET QONORDER = 700
            WHERE PARTNO = :PARTNO AND
                  QONORDER = 0 AND
                  DELIVERY_TIME = :TIME;
%SKIP;
/*****
/*
/*          PRINT ON SAMPLE ORDER
/*
/*****
%SKIP;
TPRICE2 = 700 * PRICE;
PUT FILE (PRTFILE) EDIT ('700', DESCR, QONHAND, NAME, ADR, PRICE,
      TPRICE2) (SKIP(2), COL(3), A(4), COL(14), A(10), COL(24),
      F(11), COL(39), A(15), COL(55), A(35), COL(90), F(9,2),
      COL(100), F(9,2));
%SKIP;
/*****
/*
/*          DELETE PART #321
/*
/*****
%SKIP;
      STMT = 'DELETE 321 FROM QUOTATION';
      EXEC SQL DELETE FROM QUOTATIONS
            WHERE PARTNO = 321;
      STMT = 'DELETE 321 FROM INVENTORY';
      EXEC SQL DELETE FROM INVENTORY
            WHERE PARTNO = 321;
%SKIP;
/*****
/*
/*          COMMIT CHANGES
/*
/*****
%SKIP;
      STMT = 'COMMIT WORK

```

```

EXEC SQL COMMIT WORK;
%SKIP;
/*****
/*
/*          PRINT TABLES
/*
/*
*****/
%SKIP;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE INVENTORY UPDATED ***')
(PAGE, A(45));
CALL PRINT1;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE QUOTATIONS UPDATED ***'
) (PAGE, A(45));
CALL PRINT2;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE SUPPLIERS UPDATED ***')
(PAGE, A(45));
CALL PRINT3;
GOTO DONE;
%SKIP;
/*****
/*
/*          PRINT ROUTINE FOR TABLES
/*
/*
*****/
PRINT1: PROC;
%SKIP;
EXEC SQL DECLARE C1 CURSOR FOR SELECT PARTNO, DESCRIPTION,
QONHAND
FROM INVENTORY
ORDER BY PARTNO;
PUT FILE (PRTFILE) EDIT ('PARTNO', 'DESCRIPTION', 'QONHAND')
(SKIP(2), A(6), X(3), A(11), X(3), A(7));
EXEC SQL WHENEVER NOT FOUND CONTINUE;
STMT = 'OPEN C1 - PROC PRINT1  ';
EXEC SQL OPEN C1;
DO WHILE (SQLCODE = 0);
STMT = 'FETCH C1 IN PROC PRINT1  ';
EXEC SQL FETCH C1
INTO :PARTNO, :DESCR, :QONHAND;
IF SQLCODE = 0 THEN
PUT FILE (PRTFILE) EDIT (PARTNO, DESCR, QONHAND) (SKIP,
F(6), X(3), A(10), X(5), F(6));
END;
STMT = 'CLOSE C1 IN PROC PRINT1  ';
EXEC SQL CLOSE C1;
%SKIP;
END PRINT1;
%SKIP;
PRINT2: PROC;
%SKIP;
EXEC SQL DECLARE C2 CURSOR FOR
SELECT SUPPNO, PARTNO, PRICE, DELIVERY_TIME, QONORDER
FROM QUOTATIONS
ORDER BY SUPPNO, PARTNO;
PUT FILE (PRTFILE) EDIT ('SUPPNO', 'PARTNO', 'PRICE',
'DELIVERY TIME', 'QONORDER') (SKIP(2), A(6), X(3), A(6),
X(6), A(5), X(3), A(13), X(6), A(8));
STMT = 'OPEN C2 - PROC PRINT2  ';
EXEC SQL OPEN C2;
DO WHILE (SQLCODE = 0);
STMT = 'FETCH C2 IN PROC PRINT2  ';
EXEC SQL FETCH C2
INTO :SUPPNO, :PARTNO, :PRICE, :TIME, :QONORDER;
IF SQLCODE = 0 THEN
PUT FILE (PRTFILE) EDIT (SUPPNO, PARTNO, PRICE, TIME,
QONORDER) (SKIP, F(6), X(3), F(6), X(3), F(8,2),

```

```

                X(3), F(13), X(3), F(11));
END;
STMT = 'CLOSE C2 IN PROC PRINT2  ';
EXEC SQL CLOSE C2;
%SKIP;
END PRINT2;
%SKIP;
PRINT3: PROC;
%SKIP;
EXEC SQL DECLARE C3 CURSOR FOR
SELECT SUPPNO, NAME, ADDRESS
FROM SUPPLIERS
ORDER BY SUPPNO;
PUT FILE (PRTFILE) EDIT ('SUPPNO', 'NAME', 'ADDRESS') (SKIP(2),
A(6), X(3), A(4), X(9), A(7));
STMT = 'OPEN C3 - PROC PRINT3  ';
EXEC SQL OPEN C3;
DO WHILE (SQLCODE = 0);
STMT = 'FETCH C3 IN PROC PRINT3  ';
EXEC SQL FETCH C3
INTO :SUPPNO, :NAME, :ADR;
IF SQLCODE = 0 THEN
PUT FILE (PRTFILE) EDIT (SUPPNO, NAME, ADR) (SKIP,
F(6), X(3), A(10), X(3), A(35));
END;
STMT = 'CLOSE C3 IN PROC PRINT3  ';
EXEC SQL CLOSE C3;
%SKIP;
END PRINT3;
%SKIP;
/*****
/*
/*          ROUTINE FOR HANDLING ERRORS          */
/*
/*
*****/
%SKIP;
SQLERR:
DISPLAY ('UNEXPECTED SQL ERROR RETURNED');
DISPLAY ('CHANGES WILL BE BACKED OUT');
DISPLAY ('FAILING SQL STATEMENT IS');
DISPLAY (STMT);
PUT FILE (PRTFILE) SKIP EDIT ('UNEXPECTED SQL ERROR RETURNED')
(A(29));
PUT FILE (PRTFILE) SKIP EDIT ('FAILING SQL STATEMENT IS ', STMT)
(2(A(25)));
PUT FILE (PRTFILE) SKIP EDIT ('SQLCODE: ', SQLCODE) (A(9), F(4));
PUT FILE (PRTFILE) SKIP EDIT ('SQLERRM: ', SQLERRM) (A(9), A(70));
PUT FILE (PRTFILE) SKIP EDIT ('SQLERRP: ', SQLERRP) (A(9), A(8));
PUT FILE (PRTFILE) SKIP EDIT ('SQLERRD: ', SQLERRD) (A(9), 6 F(5));
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN0: ', SQLWARN0) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN1: ', SQLWARN1) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN2: ', SQLWARN2) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN3: ', SQLWARN3) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN4: ', SQLWARN4) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN5: ', SQLWARN5) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN6: ', SQLWARN6) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN7: ', SQLWARN7) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN8: ', SQLWARN8) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN9: ', SQLWARN9) (A(10), A);
PUT FILE (PRTFILE) SKIP EDIT ('SQLWARNA: ', SQLWARNA) (A(10), A);
%SKIP;
/* IGNORE ERRORS DURING ROLLBACK TO AVOID ERROR ROUTINE LOOP          */
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK;
%SKIP;
/*****
/*

```

```
/*          PRINT END MSG AND TERMINATE PROGRAM          */
/*          */
/*****
%SKIP;
DONE: DISPLAY ('SAMPLE PROGRAM ARISPLIC COMPLETED');
      PUT FILE (PRTFILE) EDIT ('SAMPLE PROGRAM ARISPLIC COMPLETED')
          (PAGE, A(33));
%SKIP;
END SAMPP;
```



Rules for Using SQL in PL/I

This section lists, for your reference, all the rules for embedding SQL statements within a PL/I program.


Placement and Continuation of SQL Statements

The PL/I program that is to be called first must have an `OPTIONS(MAIN)` clause on the first statement. All statements in your PL/I program, including PL/I source statements and SQL statements, must be contained in columns 2 through 72 of your source deck. Normal PL/I continuation rules apply.

DBCS strings appearing in comments embedded in an SQL statement must begin and end on the same line.

Delimiting SQL Statements

Delimiters are required on all SQL statements to help SQL/DS distinguish them from regular PL/I statements. You must precede each SQL statement in your program with `EXEC SQL` and end each statement with a semicolon, as shown in the sample program. `EXEC` and `SQL` must be on the same line with only blanks separating them (no in-line comments).



Within SQL statements, comments are allowed anywhere that blanks are allowed. However, there should not be any comments within SQL statements that are dynamically defined and executed.

An SQL statement cannot be followed on the same line by another SQL statement, a normal PL/I statement, or a comment. When you preprocess a program containing such a combination, the trailing statements or comments will not appear in the `SYSPRINT` listing.

Using the INCLUDE Command

To include external secondary input, specify:

```
EXEC SQL INCLUDE text-name;
```

at the point in the source code where the secondary input is to be included. The `text-name` is the filename of a CMS file with a `PLICOPY` filetype and located on a CMS minidisk accessed by the user.

The `INCLUDE` command must be completely contained within one line. There must be no comments between the `EXEC SQL` and the semicolon (if there are, the preprocessor will not recognize the command). Additional commands cannot appear on the same line, but comments that follow the semicolon and finish on the same line are allowed.

Declaring Static External Variables

A declaration for a variable with the attributes `STATIC` and `EXTERNAL` must also have the attribute `INITIAL`. If it does not, the declaration generates a common CSECT, which SQL/DS cannot handle.

PL/I programming using “`DEFAULT RANGE (*) STATIC`” gives an error message. The SQL/DS preprocessor builds control blocks that are incompatible with this statement.

Declaring Host Variables

You must declare all PL/I variables that are to be used in SQL statements. The declarations, as shown in the sample program, must appear in a section that begins with

```
EXEC SQL BEGIN DECLARE SECTION;
```

and ends with

```
EXEC SQL END DECLARE SECTION;
```

These two statements must each be on a single line. There must be no comments between the “`EXEC SQL`” and the semicolon. (If there are, the preprocessor won't recognize the statement.)

You can have a label on the “`EXEC SQL BEGIN DECLARE SECTION;`”, but not on the “`EXEC SQL END DECLARE SECTION;`”. If you do place a label on this statement, the preprocessor does not recognize it and assumes that the special declare section hasn't ended.

When placing comments after either of these statements, make sure the comment ends on the same line. If it doesn't, PL/I compiler errors result.

PL/I `DECLARE` statements within the SQL declare section are also subject to a few rules:

- “`DCL`” or “`DECLARE`” must be the first character sequence on the line. You can, however, have a carriage control character in column 1. Otherwise, the line is ignored. (You can place in-line comments anywhere after the `DECLARE` or `DCL` keyword; and you can continue these comments over multiple lines.)
- `DECLARE` statements can be continued on additional lines, but more than one `DECLARE` statement cannot be on the same line. All `DECLARE` statements must end with a semicolon.
- Only one host variable can be declared per `DCL` or `DECLARE` statement. If multiple variables are declared, only the first variable is recognized; the others are ignored:

```
DCL AA FIXED BIN(15) INIT(7),  
    BB CHAR(7),  
    CC BINARY FLOAT(53);
```

BB and CC are ignored.

The exception is that if multiple variables have exactly the same attributes, they can be combined into a single DECLARE statement:

```
DCL (I,J,K) FIXED BIN(15) INIT(7);
```

All are recognized.

- The data type of the variable must be stated before any other attributes:

```
DECLARE XX BIN FIXED(31) STATIC;
```

Correct

```
DECLARE XX STATIC BIN FIXED(31);
```

Incorrect

- The data type can be stated in any way that is acceptable to PL/I; BIN FIXED(31), BINARY FIXED(31), and FIXED BIN(31) are all equivalent. If several variables have exactly the same attributes, you can combine them in a single DCL statement:

```
DCL (X,Y,Z) BIN FIXED;
```

- Host variables must be scalars (not arrays or structures).
- You must not declare two host variables with the same name in a single program, even if they are in different blocks or procedures. (SQL host variables can be declared once and referenced throughout the program.)
- You should not declare variables whose names begin with SQL, ARI, or RDI, because these names are reserved for SQL/DS use.
- If a host variable is used within an SQL statement, the SQL statement must be within the scope of the variable's declaration.

There can be more than one SQL declare section in a program. A host variable, however, must be declared earlier in the program than the first use of the variable in an SQL statement.

Note that other program variables can also be declared as usual outside the SQL declare section. The previous restrictions do not apply to non-SQL declarations.

Using Host Variables in SQL Statements

When you place host variables within an SQL statement, you must precede each such variable with a colon (:). The colon distinguishes the host variables from the SQL identifiers (such as PARTNO). When the same variable is used outside of an SQL statement, do not use a colon.

A host variable can represent a data value, but not an SQL identifier. For example, you cannot assign a character constant, such as 'MUSICIANS', to a host variable,

and then use that host variable in a CREATE TABLE statement to represent the table name. This pseudo-code sequence is invalid:

```
TT = 'MUSICIANS'  
CREATE TABLE :TT (NAME ...
```

Incorrect

PL/I Data Conversion Notes

Host variables must be type-compatible with the columns with which they are to be used. For example, if you want to compare a program variable with the QONHAND column of the data base, and the data type of QONHAND is INTEGER, you should declare the program variable BIN FIXED(31), BIN FIXED(15), BIN FLOAT(53), or FIXED DECIMAL(10).

SQL/DS considers the four numeric data types compatible; SQL/DS also considers the three types of character strings (fixed, varying, and long -- including strings of different declared lengths) compatible. SQL/DS also considers the three types of DBCS strings to be similarly compatible. Of course, an overflow condition may result if, for example, you assign a 31-bit integer to a 15-bit integer and the current value of the 31-bit integer is too large to fit in 15 bits. Truncation also occurs when a decimal number having a scale greater than zero is assigned to an integer. In general, overflow occurs when significant digits are lost, but truncation occurs when non-significant digits are lost.

Refer to "Data Conversion" on page 76 for a data conversion summary.

Using the Double-Byte Character Set (DBCS)

The GRAPHIC preprocessor option must be specified in order to use DBCS data.

PL/I DBCS constants are used in the SQL DECLARE section. As opposed to PL/I DBCS constants used outside of the SQL DECLARE section, these cannot span across lines.

The DBCS constant in an SQL statement embedded in a PL/I program is:

```
so'...'Gsi
```

This is the same as the DBCS constant in PL/I.

The letter G and the apostrophes (') appear inside the so/si delimiters. Therefore they are encoded as DBCS characters. Apostrophe is X'427D' and G is X'42C7'. The so and si are single-byte characters, X'0E' and X'0F'. DBCS apostrophes (X'427D') must be doubled in order to obtain a single DBCS apostrophe. As opposed to the PL/I DBCS constant, a DBCS constant in an SQL statement cannot have X'0F' as the first byte of a DBCS character.

The SQL/DS PL/I preprocessor converts PL/I format DBCS constants into SQL format DBCS constants (G'so...si') when they appear in SQL statements. This is done prior to passing the SQL statement to SQL/DS for processing. Therefore, some SQL/DS messages for incorrect syntax may refer to the SQL format of the constant, even though a PL/I format constant was coded in your program.

When the DBCS option is set to YES, SQL identifiers with DBCS characters can be used in SQL statements. For more information on SQL identifiers, see “General Rules for Naming Data Objects” on page 74.

When the DBCS option is set to YES, both character string constants in SQL statements and character string constants in PL/I statements can contain DBCS strings which are enclosed by so and si. However, no DBCS string can span across a line in the program. An apostrophe (X'7D') in a DBCS string does not terminate a character string constant and does not have to be duplicated. Therefore, note that if there is a X'7D' between so and si in a character string constant in a PL/I statement, it cannot be correctly compiled by the PL/I compiler.

Using SQL Statements in PL/I Subroutines

The first SQL statement encountered in a sequential scan of your program by the SQL/DS PL/I preprocessor results in a generated control block named SQLTIE, commonly used by internal SQL/DS code that is associated with the remaining SQL statements in your program. If your program structure involves SQL statements in multiple procedures, it is essential that you maintain a structure whereby the SQLTIE is addressable by all other SQL statement occurrences in your program.

The following is an incorrect structure:

```
A: PROC OPTIONS (MAIN) ;
    .
    .
    CALL B;
    .
    .
    CALL C;
    .
    .
B: PROC;
    .
    EXEC SQL CONNECT.....
    EXEC SQL DECLARE C1 CURSOR....
    EXEC SQL OPEN C1 ...
    .
    .
    END B;
C: PROC;
    .
    .
    EXEC SQL DECLARE C2 CURSOR....
    EXEC SQL OPEN C2 .....
    .
    .
    END C;
    .
    .
    END A;
```

The SQLTIE will be generated from the CONNECT in B, but it is not addressable from C, where other SQL statements appear. This can be solved in this case by

putting the CONNECT statement in A, where it will cause the SQLTIE to be generated at a place that is addressable by from both B and C.

SQL Statements in External Procedures

The SQL/DS PL/I preprocessor generates control blocks in your source program to communicate information to the SQL/DS system. These control blocks have the default storage class: AUTOMATIC. As a result, storage is allocated for these control blocks upon each invocation of the external procedure that contains them. Performance considerations may make it necessary for you to locate SQL statements in the main procedure of your program to avoid this the allocation time for automatic storage. This will depend upon the frequency of calls and the number of SQL statements involved, as well as the nature of your application.

SQL Error Handling

There are two ways to declare the return code structure (called SQLCA):

1. You may write:

```
EXEC SQL INCLUDE SQLCA;
```

in your source program. The SQL/DS preprocessor replaces this with the declaration of the SQLCA structure.

2. You may declare the SQLCA structure directly as shown in Figure 37.

```
DCL 1 SQLCA,  
  2 SQLCAID CHAR(8)  
  2 SQLCABC BIN FIXED(31)  
  2 SQLCODE BIN FIXED(31),  
  2 SQLERRM CHAR(70) VAR,  
  2 SQLERRP CHAR(8),  
  2 SQLERRD (6) BIN FIXED(31),  
  2 SQLWARN,  
    3 SQLWARN0 CHAR(1),  
    3 SQLWARN1 CHAR(1),  
    3 SQLWARN2 CHAR(1),  
    3 SQLWARN3 CHAR(1),  
    3 SQLWARN4 CHAR(1),  
    3 SQLWARN5 CHAR(1),  
    3 SQLWARN6 CHAR(1),  
    3 SQLWARN7 CHAR(1),  
    3 SQLWARN8 CHAR(1),  
    3 SQLWARN9 CHAR(1),  
    3 SQLWARNA CHAR(1),  
  2 SQLEXT CHAR(5);
```

Figure 37. SQLCA Structure (in PL/I)

The SQLCA must not be declared within the SQL declare section. The meanings of the fields within the SQLCA are discussed under “Error Handling” on page 202.

Dynamic SQL Statements in PL/I

You may need to declare an SQLDA structure to execute dynamically defined SQL statements. (See “Dynamically Defined Statements” on page 147.) You can have SQL/DS include the structure automatically by specifying:

```
EXEC SQL INCLUDE SQLDA;
```

in your source code, or by directly coding the structure as shown in Figure 38.

```
DCL 1 SQLDA BASED (SQLDAPTR),
    2 SQLDAID CHAR(8),
    2 SQLDABC BIN FIXED(31),
    2 SQLN BIN FIXED(15),
    2 SQLD BIN FIXED(15),
    2 SQLVAR (SQLSIZE REFER (SQLN)),
        3 SQLTYPE BIN FIXED(15),
        3 SQLLEN BIN FIXED(15),
        3 SQLDATA PTR,
        3 SQLIND PTR,
        3 SQLNAME CHAR(30) VAR;
DCL SQLSIZE BIN FIXED(15);
DCL SQLDAPTR PTR;
```

Figure 38. SQLDA Structure (in PL/I)

The SQLDA must not be declared within the SQL declare section.

In addition to the structure above, it is recommended that you declare an additional mapping for the same area. Figure 39 shows this mapping.

```
DCL 1 SQLDAX BASED (SQLDAPTR),
    2 SQLDAIDX CHAR(8),
    2 SQLDABCX BIN FIXED(31),
    2 SQLNX BIN FIXED(15),
    2 SQLDX BIN FIXED(15),
    2 SQLVARX (SQLSIZE REFER (SQLNX)),
        3 SQLTYPEX BIN FIXED(15),
        3 SQLPRCSN BIN FIXED(8),
        3 SQLSCALE BIN FIXED(8),
        3 SQLDATAAX PTR,
        3 SQLINDX PTR,
        3 SQLNAMEX CHAR(30) VAR;
```

Figure 39. SQLDAX Structure (in PL/I)

The SQLPRCSN and SQLSCALE fields of the second mapping are used when decimal data is used. The other fields in the second mapping should be ignored.

Because the PL/I SQLDA is declared as a based structure, your program can dynamically allocate an SQLDA of adequate size for use with each EXECUTE statement. For example, the code fragment below allocates an SQLDA adequate for five fields and uses it in an EXECUTE of statement S3:

```
SQLSIZE=5;
ALLOCATE SQLDA SET(SQLDAPTR);
/* Add code to set values and pointers in the SQLDA */
EXEC SQL EXECUTE S3 USING DESCRIPTOR SQLDA;
```

The statement SQLSIZE=5 determines the size of the SQLDA to be allocated by means of the PL/I REFER feature. The ALLOCATE statement allocates an SQLDA of the size desired and sets SQLDAPTR to point to it. (Before an EXECUTE statement is issued using this SQLDA, your program must fill in its contents.)

You can use a similar technique to allocate an SQLDA for use with a DESCRIBE statement. The following program fragment illustrates the use of SQLDA with DESCRIBE for three fields and a "prepared" statement S1:

```
EXEC SQL DECLARE C1 CURSOR FOR S1;
SQLSIZE = 3;
ALLOCATE SQLDA SET(SQLDAPTR);
EXEC SQL DESCRIBE S1 INTO SQLDA;
IF SQLD > SQLN THEN
  - get a bigger one
  Set SQLDATA and SQLIND
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA;
```

Data Types

Description	SQL/DS Keyword	Equivalent PL/I Declaration
A binary integer of 31 bits, plus sign.	INTEGER	BINARY FIXED(31)
A binary integer of 15 bits, plus sign.	SMALLINT	BINARY FIXED(15)
A packed decimal number, precision m, scale n ($1 \leq m \leq 15$ and $0 \leq n \leq m$). In storage the number occupies an even number of bytes up to a maximum of 8 bytes. Precision is the total number of digits. Scale is the number of those digits that are to the right of the decimal point.	DECIMAL(m[,n])	FIXED DECIMAL(m,n)
A double-precision (8- byte) floating point number, in standard System/370 floating point format.	FLOAT	BINARY FLOAT(53)
A fixed-length character string of length n where $n \leq 254$.	CHAR(n)	CHARACTER(n)
A varying-length character string of maximum length n, where $n \leq 254$.	VARCHAR(n)	CHARACTER(n) VARYING
A varying-length character string of maximum length 32767 bytes, subject to certain usage limitations.	LONG VARCHAR	CHARACTER(n) VARYING
A fixed-length DBCS character string of n DBCS characters where $n \leq 127$.	GRAPHIC(n)	GRAPHIC(n)
A varying-length DBCS character string of n DBCS characters where $n \leq 127$.	VARGRAPHIC(n)	GRAPHIC(n) VARYING
A varying-length DBCS character string of maximum length 16383, subject to certain usage limitations.	LONG VARGRAPHIC	GRAPHIC(n) VARYING

Figure 40. SQL/DS Data Types for PL/I

Note: In PL/I declarations, GRAPHIC can be abbreviated 'G'. The SQL/DS keyword GRAPHIC, however, must not be abbreviated.

Additional PL/I Program Examples

The following program is another example of how to embed SQL statements in a PL/I program.

Sometimes the BETWEEN function causes performance problems because it does not have very good information on path selection. This program shows an alternate way of performing the BETWEEN function, when these performance problems exist.


```

/*****
/*
/* THIS IS A SAMPLE PLI PROGRAM THAT PERFORMS THE BETWEEN
/* FUNCTION USING DYNAMIC SQL STATEMENTS
/*
/* WHEN THE USER CODES A STATIC SQL STATEMENT USING HOST
/* VARIABLES FOR THE BETWEEN FUNCTION, THE SQL/DS OPTIMIZER
/* DOES NOT HAVE VERY GOOD INFORMATION FOR PATH SELECTION.
/* THEREFORE, THE USER SHOULD USE SQL DYNAMIC STATEMENTS
/* TO PERFORM THIS FUNCTION.
/*
/*****
BETWDYNP: PROC OPTIONS (MAIN);
      DCL ADDR BUILTIN;
/*****
/*
/*          ESTABLISH HOST VARIABLES
/*
/*****
      EXEC SQL BEGIN DECLARE SECTION;
          DCL PARTNO BINARY FIXED (15);
          DCL DESCR CHAR(24) VARYING;
          DCL DESCR_IND BINARY FIXED (15);
          DCL QONHAND BINARY FIXED (31);
          DCL ID CHAR(8) INIT('SQLDBA ');
          DCL PASSW CHAR(8) INIT('SQLDBAPW');
          DCL ESTRING CHAR(66) VARYING;
      EXEC SQL END DECLARE SECTION;
/*          INTERNAL PROGRAM VARIABLES
          DCL P2 PIC '9999999';
          DCL P4 PIC '9999999';
/*****
/*
/*          OPEN PRINT FILE AND DISPLAY START MSG
/*
/*****
DCL PRTPFILE FILE STREAM OUTPUT PRINT;
DISPLAY ('SAMPLE PROGRAM BETWDYNP STARTED');
PUT FILE (PRTPFILE) SKIP EDIT ('SAMPLE PROGRAM BETWDYNP STARTED')
      (A(31));
/*****
/*
/*          ERROR HANDLING
/*
/*****
      EXEC SQL INCLUDE SQLCA;
/* THIS PROGRAM WILL IGNORE WARNING AS THEY WILL NOT AFFECT RESULTS */
      EXEC SQL WHENEVER SQLWARNING GOTO SQLERR;
      EXEC SQL WHENEVER SQLERROR GOTO SQLERR;
      EXEC SQL WHENEVER NOT FOUND GOTO END_DATA;
/*****
/*
/*          START PROGRAM
/*
/*****
      EXEC SQL CONNECT :ID IDENTIFIED BY :PASSW;
GET_RANGE_VALUES:
/*****
/*
/* THERE ARE MANY WAYS TO GET THE RANGE VALUES FOR
/* THE BETWEEN FUNCTION INTO YOUR PROGRAM. THIS PROGRAM
/* WILL SIMPLY MOVE VALUES INTO THE DATA STRING OF THE
/* SQL COMMAND.

```

```

/*
/*****
P2 = 75;
P4 = 100;
ESTRING = 'SELECT * FROM INVENTORY WHERE QONHAND BETWEEN '
          || P2 || ' AND ' || P4;
PREPARE_REQUEST:
EXEC SQL PREPARE STAT1 FROM :ESTRING;
DECLARE_CURSOR:
EXEC SQL DECLARE C1 CURSOR FOR STAT1;
OPEN_CURSOR:
EXEC SQL OPEN C1;
PUT FILE(PRTFILE) EDIT ('PARTNO','DESCRIPTION','QONHAND')
          (SKIP(2),A(6),X(3),A(11),X(16),A(7));
GET_NEXT:
EXEC SQL FETCH C1
          INTO :PARTNO, :DESCR:DESCR_IND, :QONHAND;
IF LENGTH(DESCR)=0 THEN DESCR='ZERO LENGTH';
IF DESCR_IND<0 THEN DESCR='NULL DATA';
PUT FILE(PRTFILE) EDIT (PARTNO, DESCR, QONHAND)
          (SKIP,F(6),X(3),A(24),X(4),F(6));
GO TO GET_NEXT;

/*****
/*
/*          ROUTINE FOR HANDLING ERRORS          */
/*
/*****
SQLERR:
DISPLAY ('UNEXPECTED SQL ERROR RETURNED');
DISPLAY ('CHANGES WILL BE BACKED OUT');
DISPLAY ('FAILING SQL STATEMENT IS');
DISPLAY (STMT);
PUT FILE(PRTFILE) SKIP EDIT ('UNEXPECTED SQL ERROR RETURNED')
          (A(29));
PUT FILE(PRTFILE) SKIP EDIT ('SQLCODE: ',SQLCODE) (A(9),F(4));
PUT FILE(PRTFILE) SKIP EDIT ('SQLERRM: ',SQLERRM) (A(9),A(70));
PUT FILE(PRTFILE) SKIP EDIT ('SQLERRP: ',SQLERRP) (A(9),A(8));
PUT FILE(PRTFILE) SKIP EDIT ('SQLERRD: ',SQLERRD) (A(9),6 F(5));
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN0: ',SQLWARN0) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN1: ',SQLWARN1) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN2: ',SQLWARN2) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN3: ',SQLWARN3) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN4: ',SQLWARN4) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN5: ',SQLWARN5) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN6: ',SQLWARN6) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN7: ',SQLWARN7) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN8: ',SQLWARN8) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARN9: ',SQLWARN9) (A(10),A);
PUT FILE(PRTFILE) SKIP EDIT ('SQLWARNA: ',SQLWARNA) (A(10),A);
/* IGNORE ERRORS DURING ROLLBACK TO AVOID ERROR ROUTINE LOOP */
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK;
/*****
/*
/*          PRINT END MSG AND TERMINATE PROGRAM          */
/*
/*****
END_DATA: DISPLAY ('SAMPLE PROGRAM BETWDYNP COMPLETED');
          PUT FILE(PRTFILE) EDIT ('SAMPLE PROGRAM BETWDYNP COMPLETED')
          (PAGE, A(33));
END BETWDYNP;

```

The following program is an example of dynamic SQL statements which use the SQLDA:

```

/*****/
/*
/* THIS IS A SAMPLE PL/I PROGRAM WHICH SHOWS HOW TO USE
/* DYNAMIC SQL STATEMENTS WHICH USE THE SQLDA.
/*
/* THIS SAMPLE PROGRAM IS AN IMPLEMENTATION OF THE
/* SELECT DESCRIPTION,QONHAND FROM INVENTORY WHERE PARTNO = ?
/* QUERY THAT IS DESCRIBED UNDER DYNAMICALLY DEFINED STATEMENTS,
/* PARAMETERIZED QUERIES UNDER CODING THE PROGRAM IN CHAPTER 2
/*
/*****/
SAMPDYNP: PROC OPTIONS (MAIN);
        DCL ADDR BUILTIN;
        DCL PLIRETV BUILTIN;
/*****/
/*
/*          SQLDA DEFINITIONS
/*
/*****/
        EXEC SQL INCLUDE SQLDA;
        DCL 1 SQLDA1 BASED(SQLDAPTR1),
            2 SQLDAID1 CHAR(8),
            2 SQLDABC1 BIN FIXED(31),
            2 SQLN1 BIN FIXED,
            2 SQLD1 BIN FIXED,
            2 SQLVAR1(SQLSIZE1 REFER (SQLN1)),
                3 SQLTYPE1 BIN FIXED,
                3 SQLLEN1 BIN FIXED,
                3 SQLDATA1 PTR,
                3 SQLIND1 PTR,
                3 SQLNAME1 CHAR(30) VAR;
        DCL SQLSIZE1 BIN FIXED;
        DCL SQLDAPTR1 PTR;
/*****/
/*
/*          ERROR HANDLING
/*
/*****/
        EXEC SQL INCLUDE SQLCA;
        EXEC SQL WHENEVER SQLERROR GOTO SQLERR;
        EXEC SQL WHENEVER SQLWARNING GOTO SQLERR;
        EXEC SQL WHENEVER NOT FOUND GOTO END_DATA;
/*****/
/*
/*          DECLARE HOST VARIABLES
/*
/*****/
        EXEC SQL BEGIN DECLARE SECTION;
        DCL STRING1 CHAR(100) VAR;
        DCL STRING2 CHAR(100) VAR;
        DCL ID CHAR(8) INIT('SQLDBA ');
        DCL PASSW CHAR(8) INIT('SQLDBAPW');
        EXEC SQL END DECLARE SECTION;
/*****/
/*
/*          DECLARE OTHER PROGRAM VARIABLES
/*
/*****/
        DCL POINTER PTR;
        DCL PARTNO FIXED BINARY(15) BASED(POINTER);

```

```

DCL DESCR CHAR(24) VAR BASED(POINTER);
DCL DESCR_IND BINARY FIXED(15) BASED(POINTER);
DCL QONHAND FIXED BINARY(31) BASED(POINTER);
DCL QONHAND_IND BINARY FIXED(15) BASED(POINTER);
/*****
/*
/*      OPEN PRINT FILE AND DISPLAY START MSG
/*
/*
/*****
DCL PRTFILE FILE STREAM OUTPUT PRINT;
DISPLAY ('SAMPLE PROGRAM SAMPDYNP STARTED');
PUT FILE (PRTFILE) SKIP EDIT ('SAMPLE PROGRAM SAMPDYNP STARTED')
      (A(31));
/*****
/*
/*      START PROGRAM
/*
/*
/*****
EXEC SQL CONNECT :ID IDENTIFIED BY :PASSW;
STRING1 = 'SELECT DESCRIPTION, QONHAND FROM SQLDBA.INVENTORY WHERE
PARTNO = ?';
EXEC SQL PREPARE STAT1 FROM :STRING1;
SQLSIZE = 2;
ALLOCATE SQLDA SET(SQLDAPTR);
SQLN = 2;
EXEC SQL DESCRIBE STAT1 INTO SQLDA;
ALLOCATE DESCR SET(POINTER);
SQLDATA(1) = POINTER;
ALLOCATE DESCR_IND SET(POINTER);
SQLIND(1) = POINTER;
ALLOCATE QONHAND SET(POINTER);
SQLDATA(2) = POINTER;
ALLOCATE QONHAND_IND SET(POINTER);
SQLIND(2) = POINTER;
EXEC SQL DECLARE C1 CURSOR FOR STAT1;
STRING2 = 'SELECT PARTNO FROM SQLDBA.INVENTORY';
SQLSIZE1 = 1;
ALLOCATE SQLDA1 SET (SQLDAPTR1);
SQLN1 = 1;
EXEC SQL PREPARE STATEMENT1 FROM :STRING2;
EXEC SQL DESCRIBE STATEMENT1 INTO SQLDA1;
ALLOCATE PARTNO SET (POINTER);
SQLDATA1(1)=POINTER;
/*****
/*
/*      THERE ARE MANY WAYS TO GET THE PARTNO INTO YOUR
/*      PROGRAM. THIS PROGRAM WILL SIMPLY MOVE A VALUE
/*      INTO THE PARTNO VARIABLE.
/*
/*
/*****
PARTNO = 207;
EXEC SQL OPEN C1 USING DESCRIPTOR SQLDA1;
PUT FILE(PRTFILE) EDIT ('PARTNO','DESCRIPTION','QONHAND')
      (SKIP(2),A(6),X(3),A(11),X(16),A(7));
GET_NEXT:
EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA;
IF LENGTH(SQLDATA(1) -> DESCR)=0
  THEN SQLDATA(1) -> DESCR = 'ZERO LENGTH';
IF SQLIND(1) -> DESCR_IND < 0
  THEN SQLDATA(1) -> DESCR = 'NULL DATA';
IF SQLIND(2) -> QONHAND_IND = 0
  THEN PUT FILE (PRTFILE) EDIT
      (SQLDATA1(1) -> PARTNO, SQLDATA(1) -> DESCR,
      SQLDATA(2) -> QONHAND)
      (SKIP,F(6),X(3),A(24),X(4),F(6));
ELSE PUT FILE (PRTFILE) EDIT
      (SQLDATA1(1) -> PARTNO, SQLDATA(1) -> DESCR,

```

```

                ' NULL')
                (SKIP,F(6),X(3),A(24),X(4),A(6));
GO TO GET_NEXT;
/*****
/*
/*          ROUTINE FOR HANDLING ERRORS          */
/*
/*
*****/
SQLERR:
  DISPLAY ('UNEXPECTED SQL ERROR RETURNED');
  DISPLAY ('CHANGES WILL BE BACKED OUT');
  PUT FILE (PRTFILE) SKIP EDIT ('UNEXPECTED SQL ERROR RETURNED')
    (A(29));
  PUT FILE (PRTFILE) SKIP EDIT ('SQLCODE: ',SQLCODE) (A(9),F(4));
  PUT FILE (PRTFILE) SKIP EDIT ('SQLERRM: ',SQLERRM) (A(9),A(70));
  PUT FILE (PRTFILE) SKIP EDIT ('SQLERRP: ',SQLERRP) (A(9),A(8));
  PUT FILE (PRTFILE) SKIP EDIT ('SQLERRD: ',SQLERRD) (A(9),6 F(5));
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN0: ',SQLWARN0) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN1: ',SQLWARN1) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN2: ',SQLWARN2) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN3: ',SQLWARN3) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN4: ',SQLWARN4) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN5: ',SQLWARN5) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN6: ',SQLWARN6) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN7: ',SQLWARN7) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN8: ',SQLWARN8) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN9: ',SQLWARN9) (A(10),A);
  PUT FILE (PRTFILE) SKIP EDIT ('SQLWARNA: ',SQLWARNA) (A(10),A);
/* IGNORE ERRORS DURING ROLLBACK TO AVOID ERROR ROUTINE LOOP */
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK;
/*****
/*
/*          PRINT END MSG AND TERMINATE PROGRAM          */
/*
/*
*****/
END_DATA:
  DISPLAY ('SAMPLE PROGRAM SAMPDYNP COMPLETED');
  PUT FILE (PRTFILE) EDIT ('SAMPLE PROGRAM SAMPDYNP COMPLETED')
    (PAGE, A(33));
  EXEC SQL CLOSE C1;
%SKIP;
END SAMPDYNP;

```

The following is an example of how to mix isolation levels in a PL/I program:

```

/*****
/*
/* SAMPLE PROGRAM      ARISAMPP
/*
/* PURPOSE:           THIS PROGRAM SERVES TWO PURPOSES:
/*                     1. IT IS AN EXAMPLE FOR HOW TO IMBED SQL
/*                       STATEMENTS IN A PL/1 PROGRAM.
/*                     2. IT CAN BE USED TO TEST SOME BASIC SQL
/*                       FUNCTIONS FROM AN APPLICATION PROGRAM.
/*
/* DESCRIPTION:       THIS PROGRAM GENERATES A SAMPLE ORDER FOR
/*                     THE PARTS 302 AND 311 IF QONHAND IS LESS
/*                     THAN 1000 AND 700 RESPECTIVELY, AND IF
/*                     QONORDER IS ZERO.  THE TABLE QUOTATIONS
/*                     IS UPDATED ACCORDINGLY.  PART 302 IS ORDERED
/*                     FROM THE COMPANY THAT SELLS IT FOR THE
/*                     LOWEST PRICE, PART 311 FROM THE COMPANY WITH
/*                     THE SHORTEST DELIVERY TIME.
/*
/*                     AT THE END OF THE PROGRAM PART 321 IS DELETED
/*                     FROM THE DATA BASE.
/*
/* PREREQUISITE:      A SUCCESSFUL EXECUTION OF PROGRAM ARISAMDB.
/*
/* OUTPUT PRODUCED:   1. AN EXECUTION BEGIN AND END MESSAGE IS
/*                     PRINTED AT BEGIN AND END OF PROGRAM
/*                     EXECUTION.
/*                     2. ALL TABLES ARE PRINTED WITH THEIR ORIGI-
/*                     NAL CONTENTS.
/*                     3. A SAMPLE ORDER IS PRINTED.
/*                     4. THE CONTENTS OF THE TABLES ARE PRINTED
/*                     AFTER ALL UPDATES AND DELETES ARE MADE.
/*                     5. UNEXPECTED RETURN CODE:
/*                     AN ERROR MESSAGE IS ISSUED TOGETHER WITH
/*                     THE SQLCA-INFORMATION AND CHANGES BACKED
/*                     OUT.
/*
/*****
SAMPP: PROC OPTIONS (MAIN);
%SKIP;
/*****
/*
/*          ESTABLISH HOST VARIABLES
/*
/*
/*****
%SKIP;
EXEC SQL BEGIN DECLARE SECTION;
%SKIP;
DCL PARTNO DEC FIXED(6);
/* THE DESCRIPTION COLUMN IN THE INVENTORY TABLE IS VARCHAR(24)
/* SQL/DS WILL CONVERT IT TO FIXED LENGTH AND TRUNCATE LONGER
/* DATA WHENEVER THE DESCR HOST VARIABLE IS USED.
DCL DESCR CHAR(10);
DCL QONHAND DEC FIXED(11);
DCL SUPPNO DEC FIXED(6);
DCL NAME CHAR(15);
DCL ADR CHAR(35);
DCL TIME DEC FIXED(6);
DCL QONORDER DEC FIXED(11);
DCL PRICE DEC FIXED(5,2);
DCL ID CHAR(8) INIT('SQLDBA ');
DCL PASSW CHAR(8) INIT('SQLDBAPW');

```

```

        DCL SQLISL CHAR(1) INIT('R');
%SKIP; EXEC SQL END DECLARE SECTION;
%SKIP;
/*          INTERNAL PROGRAM VARIABLES          */
%SKIP;
        DCL STMT CHAR(25);
        DCL TPRICE1 DEC FIXED(9,2);
        DCL TPRICE2 DEC FIXED(9,2);
%SKIP;
/*****
/*
/*          OPEN PRINT FILE AND DISPLAY START MSG          */
/*
*****/
%SKIP;
DCL PRTFILE FILE STREAM OUTPUT PRINT ENVIRONMENT (MEDIUM(SYSLST,1403)
        F BUFFERS(1));
DISPLAY ('SAMPLE PROGRAM ARISAMPP STARTED');
PUT FILE (PRTFILE) SKIP EDIT ('SAMPLE PROGRAM ARISAMPP STARTED')
        (A(33));
%SKIP;
/*****
/*
/*          ERROR HANDLING          */
/*
*****/
%SKIP;
        EXEC SQL INCLUDE SQLCA;
/* THIS PROGRAM WILL IGNORE WARNING AS THEY WILL NOT AFFECT RESULTS */
        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER SQLERROR GOTO SQLERR;
        EXEC SQL WHENEVER NOT FOUND GOTO SQLERR;
%SKIP;
/*****
/*
/*          START PROGRAM          */
/*
*****/
%SKIP;
        STMT = 'EXEC SQL CONNECT          ';
        EXEC SQL CONNECT :ID IDENTIFIED BY :PASSW;
%SKIP;
/*****
/*
/*          PRINT TABLES          */
/*
*****/
%SKIP;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE INVENTORY UNCHANGED ***
') (PAGE, A(45));
SQLISL = 'C';
CALL PRINT1;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE QUOTATIONS UNCHANGED **
*') (PAGE, A(45));
SQLISL = 'R';
CALL PRINT2;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE SUPPLIERS UNCHANGED ***
') (PAGE, A(45));
SQLISL = 'C';
CALL PRINT3;
%SKIP;
/*****
/*
/*          FIND MINIMUM PRICE FOR PART #302          */
/*
*****/

```

```

/* THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM PRICE OF ALL */
/* OCCURRENCES OF PART #302 WITH QONHAND LESS THAN 1000, AND */
/* QONORDER 0. AS PRICE IS A COLUMN IN QUOTATIONS, AND QONHAND A */
/* COLUMN IN INVENTORY, THE TWO TABLES HAVE TO BE LINKED VIA A JOIN */
/* BETWEEN THE PART NUMBERS IN INVENTORY AND THOSE IN QUOTATIONS. */
/* NO CURSOR IS USED BECAUSE THE STATEMENT IS EXPECTED TO RETURN */
/* ONLY ONE ROW. */
/*
/*****
%SKIP;
      STMT = 'SELECT MIN(PRICE) FOR 302';
      EXEC SQL SELECT MIN(PRICE)
            INTO :PRICE
            FROM INVENTORY, QUOTATIONS
            WHERE INVENTORY.PARTNO = 302 AND
                  QONHAND < 1000 AND
                  INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
                  QONORDER = 0;

%SKIP;
/*****
/*
/*      RETRIEVE DATA FOR ORDER AND
/*      UPDATE QONORDER FOR PART #302, TABLE QUOTATIONS
/*
/*****
%SKIP;
      TPRICE1 = 1000 * PRICE;
      STMT = 'SELECT FOR PART #302      ';
      EXEC SQL SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND,
            PRICE, NAME, ADDRESS, QUOTATIONS.SUPPNO
            INTO :PARTNO, :DESCR, :QONHAND, :PRICE, :NAME, :ADR,
            :SUPPNO
            FROM INVENTORY, QUOTATIONS, SUPPLIERS
            WHERE INVENTORY.PARTNO = 302 AND
                  INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
                  PRICE = :PRICE AND
                  QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO;
      STMT = 'UPDATE QUOTATIONS      ';
      EXEC SQL UPDATE QUOTATIONS SET QONORDER = 1000
            WHERE PARTNO = :PARTNO AND
                  QONORDER = 0 AND
                  PRICE = :PRICE AND
                  SUPPNO = :SUPPNO;

%SKIP;
/*****
/*
/*      PRINT SAMPLE ORDER WITH RESULTS OF PART #302
/*
/*****
%SKIP;
PUT FILE (PRTFILE) EDIT ('SAMPLE ORDER') (PAGE, X(30), A(12));
PUT FILE (PRTFILE) EDIT ('NUMBER OF', 'DESCRIPTION', 'QUANTITY',
      'COMPANY NAME', 'COMPANY ADDRESS', 'PRICE PER', 'TOTAL')
      (SKIP(2), COL(2), A(9), COL(14), A(11), COL(28), A(8),
      COL(39), A(12), COL(55), A(15), COL(92), A(9), COL(102),
      A(5));

%SKIP;
PUT FILE (PRTFILE) EDIT ('PARTS', 'ON HAND', 'UNIT', 'COSTS')
      (COL(4), A(5), COL(28), A(7), COL(94), A(4), COL(102), A(5));

%SKIP;
PUT FILE (PRTFILE) EDIT ('1000', DESCR, QONHAND, NAME, ADR, PRICE,
      TPRICE1) (SKIP(2), COL(3), A(4), COL(14), A(10), COL(24),
      F(11), COL(39), A(15), COL(55), A(35), COL(90), F(9,2),
      COL(100), F(9,2));

%SKIP;
/*****
/*

```



```

/*          FIND MINIMUM DELIVERY TIME FOR PART #311          */
/*          */
/* THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM DELIVERY */
/* TIME OF ALL OCCURENCES OF PART #311 WITH QONHAND LESS THAN 700 */
/* AND QONORDER 0. AS DELIVERY TIME IS A COLUMN IN QUOTATIONS */
/* AND QONHAND A COLUMN IN INVENTORY, THE TWO TABLES HAVE TO BE */
/* LINKED VIA A JOIN BETWEEN THE PART NUMBERS IN QUOTATIONS AND */
/* THOSE IN INVENTORY.                                         */
/*          */
/*****/
%SKIP;
      STMT = 'SELECT MIN(DELIVERY_TIME)';
      EXEC SQL SELECT MIN(DELIVERY_TIME)
      INTO :TIME
      FROM INVENTORY, QUOTATIONS
      WHERE INVENTORY.PARTNO = 311 AND
            QONHAND < 700 AND
            INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
            QONORDER = 0;

%SKIP;
/*****/
/*          RETRIEVE DATA OF PART 311 FOR THE ORDER          */
/*          AND                                               */
/*          UPDATE QONORDER FOR PART #311                    */
/*          */
/*****/
%SKIP;
      STMT = 'SELECT FOR PART #311          ';
      EXEC SQL SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND,
      PRICE, NAME, ADDRESS
      INTO :PARTNO, :DESCR, :QONHAND, :PRICE, :NAME, :ADR
      FROM INVENTORY, QUOTATIONS, SUPPLIERS
      WHERE INVENTORY.PARTNO = 311 AND
            INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
            DELIVERY_TIME = :TIME AND
            QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO;
      STMT = 'UPDATE QUOTATIONS          ';
      EXEC SQL UPDATE QUOTATIONS SET QONORDER = 700
      WHERE PARTNO = :PARTNO AND
            QONORDER = 0 AND
            DELIVERY_TIME = :TIME;

%SKIP;
/*****/
/*          PRINT ON SAMPLE ORDER                            */
/*          */
/*****/
%SKIP;
TPRICE2 = 700 * PRICE;
PUT FILE (PRTFILE) EDIT ('700', DESCR, QONHAND, NAME, ADR, PRICE,
TPRICE2) (SKIP(2), COL(3), A(4), COL(14), A(10), COL(24),
F(11), COL(39), A(15), COL(55), A(35), COL(90), F(9,2),
COL(100), F(9,2));

%SKIP;
/*****/
/*          DELETE PART #321                                */
/*          */
/*****/
%SKIP;
      STMT = 'DELETE 321 FROM QUOTATION';
      EXEC SQL DELETE FROM QUOTATIONS
      WHERE PARTNO = 321;
      STMT = 'DELETE 321 FROM INVENTORY';
      EXEC SQL DELETE FROM INVENTORY

```

```

WHERE PARTNO = 321;
%SKIP;
/*****
/*
/*          COMMIT CHANGES
/*
/*
/*****
%SKIP;
      STMT = 'COMMIT WORK
      EXEC SQL COMMIT WORK;
%SKIP;
/*****
/*
/*          PRINT TABLES
/*
/*
/*****
%SKIP;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE INVENTORY UPDATED ***')
(PAGE, A(45));
CALL PRINT1;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE QUOTATIONS UPDATED ***'
) (PAGE, A(45));
CALL PRINT2;
PUT FILE (PRTFILE) EDIT ('*** PRINTOUT OF TABLE SUPPLIERS UPDATED ***')
(PAGE, A(45));
CALL PRINT3;
GOTO DONE;
%SKIP;
/*****
/*
/*          PRINT ROUTINE FOR TABLES
/*
/*
/*****
PRINT1: PROC;
%SKIP;
      EXEC SQL DECLARE C1 CURSOR FOR SELECT PARTNO, DESCRIPTION,
      QONHAND
      INTO :PARTNO, :DESCR, :QONHAND
      FROM INVENTORY
      ORDER BY PARTNO;
      PUT FILE (PRTFILE) EDIT ('PARTNO', 'DESCRIPTION', 'QONHAND')
      (SKIP(2), A(6), X(3), A(11), X(3), A(7));
      EXEC SQL WHENEVER NOT FOUND CONTINUE;
      STMT = 'OPEN C1 - PROC PRINT1  ';
      EXEC SQL OPEN C1;
      DO WHILE (SQLCODE = 0);
      STMT = 'FETCH C1 IN PROC PRINT1  ';
      EXEC SQL FETCH C1;
      IF SQLCODE = 0 THEN
      PUT FILE (PRTFILE) EDIT (PARTNO, DESCR, QONHAND) (SKIP,
      F(6), X(3), A(10), X(5), F(6));
      END;
      STMT = 'CLOSE C1 IN PROC PRINT1  ';
      EXEC SQL CLOSE C1;
%SKIP;
END PRINT1;
%SKIP;
PRINT2: PROC;
%SKIP;
      EXEC SQL DECLARE C2 CURSOR FOR
      SELECT SUPPNO, PARTNO, PRICE, DELIVERY_TIME, QONORDER
      INTO :SUPPNO, :PARTNO, :PRICE, :TIME, :QONORDER
      FROM QUOTATIONS
      ORDER BY SUPPNO, PARTNO;
      PUT FILE (PRTFILE) EDIT ('SUPPNO', 'PARTNO', 'PRICE',
      'DELIVERY TIME', 'QONORDER') (SKIP(2), A(6), X(3), A(6),
      X(6), A(5), X(3), A(13), X(6), A(8));

```

```

        STMT = 'OPEN C2 - PROC PRINT2      ';
        EXEC SQL OPEN C2;
        DO WHILE (SQLCODE = 0);
            STMT = 'FETCH C2 IN PROC PRINT2  ';
            EXEC SQL FETCH C2;
            IF SQLCODE = 0 THEN
                PUT FILE (PRTFILE) EDIT (SUPPNO, PARTNO, PRICE, TIME,
                    QONORDER) (SKIP, F(6), X(3), F(6), X(3), F(8,2),
                    X(3), F(13), X(3), F(11));
            END;
        STMT = 'CLOSE C2 IN PROC PRINT2    ';
        EXEC SQL CLOSE C2;

%SKIP;
END PRINT2;
%SKIP;
PRINT3: PROC;
%SKIP;

        EXEC SQL DECLARE C3 CURSOR FOR
            SELECT SUPPNO, NAME, ADDRESS
                INTO :SUPPNO, :NAME, :ADR
                FROM SUPPLIERS
                ORDER BY SUPPNO;
        PUT FILE (PRTFILE) EDIT ('SUPPNO', 'NAME', 'ADDRESS') (SKIP(2),
            A(6), X(3), A(4), X(9), A(7));
        STMT = 'OPEN C3 - PROC PRINT3      ';
        EXEC SQL OPEN C3;
        DO WHILE (SQLCODE = 0);
            STMT = 'FETCH C3 IN PROC PRINT3  ';
            EXEC SQL FETCH C3;
            IF SQLCODE = 0 THEN
                PUT FILE (PRTFILE) EDIT (SUPPNO, NAME, ADR) (SKIP,
                    F(6), X(3), A(10), X(3), A(35));
            END;
        STMT = 'CLOSE C3 IN PROC PRINT3    ';
        EXEC SQL CLOSE C3;

%SKIP;
END PRINT3;
%SKIP;
/*****
/*
/*          ROUTINE FOR HANDLING ERRORS          */
/*
/*
*****/
%SKIP;
SQLERR:
        DISPLAY ('UNEXPECTED SQL ERROR RETURNED');
        DISPLAY ('CHANGES WILL BE BACKED OUT');
        DISPLAY ('FAILING SQL STATEMENT IS');
        DISPLAY (STMT);
        PUT FILE (PRTFILE) SKIP EDIT ('UNEXPECTED SQL ERROR RETURNED')
            (A(29));
        PUT FILE (PRTFILE) SKIP EDIT ('FAILING SQL STATEMENT IS ', STMT)
            (2(A(25)));
        PUT FILE (PRTFILE) SKIP EDIT ('SQLCODE: ', SQLCODE) (A(9), F(4));
        PUT FILE (PRTFILE) SKIP EDIT ('SQLERRM: ', SQLERRM) (A(9), A(70));
        PUT FILE (PRTFILE) SKIP EDIT ('SQLERRP: ', SQLERRP) (A(9), A(8));
        PUT FILE (PRTFILE) SKIP EDIT ('SQLERRD: ', SQLERRD) (A(9), 6 F(5));
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN0: ', SQLWARN0) (A(10), A);
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN1: ', SQLWARN1) (A(10), A);
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN2: ', SQLWARN2) (A(10), A);
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN3: ', SQLWARN3) (A(10), A);
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN4: ', SQLWARN4) (A(10), A);
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN5: ', SQLWARN5) (A(10), A);
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN6: ', SQLWARN6) (A(10), A);
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN7: ', SQLWARN7) (A(10), A);
        PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN8: ', SQLWARN8) (A(10), A);

```

```

    PUT FILE (PRTFILE) SKIP EDIT ('SQLWARN9: ',SQLWARN9) (A(10),A);
    PUT FILE (PRTFILE) SKIP EDIT ('SQLWARNA: ',SQLWARNA) (A(10),A);
%SKIP;
/* IGNORE ERRORS DURING ROLLBACK TO AVOID ERROR ROUTINE LOOP      */
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK;
%SKIP;
/*****
/*
/*          PRINT END MSG AND TERMINATE PROGRAM                    */
/*
*****/
%SKIP;
DONE: DISPLAY ('SAMPLE PROGRAM ARISAMPP COMPLETED');
      PUT FILE (PRTFILE) EDIT ('SAMPLE PROGRAM ARISAMPP COMPLETED')
          (PAGE, A(33));
%SKIP;
END SAMPP;
/*
  CLOSE SYSPCH,D
ASSGN SYSIPT,DISK,VOL=SQLWK1,SHR
// OPTION CATAL
  PHASE ARISAMPP,S
// EXEC PLIOPT,SIZE=800K
  INCLUDE ARIPRDID
// EXEC LNKEDT
/*
CLOSE SYSIPT,C
// EXEC PGM=ARISAMPP,SIZE=800K
/*
// EXEC MAINT
  DELETC ARISAMPP
/*
// ASSGN SYSIN,C
/&

```


Appendix D. COBOL Considerations

This appendix contains a sample program that illustrates the use of SQL within COBOL. Following the sample are specific rules for using SQL in COBOL.

ARISCBLC -- COBOL Sample Program

ARISCBLC is a COBOL sample program that is provided with SQL/DS. The source code of this program begins on the next page. You can learn most of the rules for using SQL within COBOL just by scanning through the program. Here is a summary by COBOL Divisions:

- Identification and Environment Divisions

You don't have to do anything different in either of these divisions for SQL/DS applications.

- Data Division

In the Data Division of any COBOL SQL/DS application, you must declare all host variables and the SQLCA structure.

Notice that the SQL statements (BEGIN DECLARE SECTION, END DECLARE SECTION, and INCLUDE SQLCA) are preceded by "EXEC SQL" and followed by "END-EXEC". These delimiters help SQL/DS distinguish SQL statements from regular COBOL code. Note also that all SQL statements must be placed in columns 12 to 72.

The only SQL statements allowed in the Data Division are those shown in the sample programs and the INCLUDE command (not shown); all others must be placed in the Procedure Division.

The COBOL PICTURE clauses for the host variables were determined by referring to Figure 42 on page 417. That figure gives the COBOL representation for each of the ten SQL/DS data types. When you are coding your own applications you'll need to obtain the data types of the columns that your host variables interact with. This can be done either by consulting the person who created the table, or by querying the SQL/DS catalogs. The SQL/DS catalogs are described in the *SQL/Data System Planning and Administration for VM/SP* manual.

- Procedure Division

A **WHENEVER** statement should be coded to provide for error handling, and the program must connect to SQL/DS.

```

*****
*
* SAMPLE PROGRAM FOR VM/SP  ARISCBLC
*
* PURPOSE:
*
*     THE PURPOSE OF THIS SAMPLE PROGRAM IS TWOFOLD.
*
*     1.  IT IS AN EXAMPLE FOR HOW TO IMBED SQL
*         STATEMENTS IN A COBOL APPLICATION PROGRAM.
*
*     2.  IT CAN BE USED TO TEST SOME BASIC SQL
*         FUNCTIONS FROM AN APPLICATION PROGRAM.
*
* DESCRIPTION:
*
*     THIS PROGRAM GENERATES A SAMPLE ORDER FOR PARTS
*     310 AND 316 IF QONHAND IS LESS THAN 1000 AND 700
*     RESPECTIVELY AND IF QONORDER IS ZERO.  THE TABLE
*     QUOTATIONS IS UPDATED ACCORDINGLY.  PART 310 IS
*     ORDERED FROM THE COMPANY THAT SELLS FOR THE LOWEST
*     PRICE, PART 316 FROM THE COMPANY WITH THE SHORTEST
*     DELIVERY TIME.
*
*     AT THE END OF THE PROGRAM PART 322 IS DELETED FROM
*     THE DATA BASE.
*
* PREREQUISITE:
*
*     THE SQL/DS SAMPLE TABLES MUST BE CREATED AND LOADED.
*
* OUTPUT PRODUCED:
*
*     CONSOLE:  1.  AN EXECUTION BEGIN AND END MESSAGE
*                 IS DISPLAYED AT THE BEGIN AND END
*                 OF PROGRAM EXECUTION.
*               2.  UNEXPECTED RETURN CODES PRODUCE AN
*                 ERROR MESSAGE.
*
*     SYSPRINT: 1.  PRINTOUT OF ALL TABLES BEFORE CHANGES.
*                 2.  SAMPLE ORDER IS PRODUCED.
*                 3.  PRINTOUT OF ALL TABLES AFTER CHANGES.
*                 4.  UNEXPECTED RETURN CODES PRODUCE AN
*                     ERROR MESSAGE WITH THE SQLCA STRUCTURE*
*                     PRINTED OUT AND CHANGES BACKED OUT.
*
*****

```

IDENTIFICATION DIVISION.

PROGRAM-ID. ARISCBLC.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SPECIAL-NAMES.

CO1 IS NEWPAGE.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT OUT-FILE ASSIGN TO UT-2400-S-OUT-FILE.

DATA DIVISION.

FILE SECTION.

FD OUT-FILE

LABEL RECORDS ARE OMITTED

DATA RECORD IS PRINT-OUT.

01 PRINT-OUT PICTURE X(130).

WORKING-STORAGE SECTION.

* HOST VARIABLE DECLARATION SECTION *

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

77 PARTNO PIC S9(6) COMP.
77 QONHAND PIC S9(9) COMP.
77 SUPPNO PIC S9(6) COMP.
77 TIMEX PIC S9(4) COMP.
77 QONORDER PIC S9(9) COMP.
77 PRICE PIC S9(9)V9(2) COMP-3.
77 TTL-PRICE PIC S9(9)V9(2) COMP-3.

* THE DESCRIPTION COLUMN IN THE INVENTORY TABLE IS DEFINED AS
* VARCHAR(24). SQL/DS WILL CONVERT DATA TO FIXED LENGTH AND
* TRUNCATE IT WHENEVER THE DESCR HOST VARIABLE IS USED.

01 DESCR PIC X(10) VALUE SPACES.
01 NAME PIC X(15) VALUE SPACES.
01 USERID PIC X(8) VALUE 'SQLDBA'.
01 PASSW PIC X(8) VALUE 'SQLDBAPW'.

01 ADDR.
49 ADDRLEN PIC S9(4) COMP VALUE +35.
49 ADDRVAL PIC X(35) VALUE SPACES.

EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL INCLUDE SQLCA END-EXEC.

* PROGRAM VARIABLE DECLARATION SECTION *

01 STEP-DENOTER PIC X(50) VALUE SPACES.

01 DECODED-SQLCODE PIC -----999.

01 ARRAY-SQLERRD.
02 DECODED-SQLERRD PIC -----999 OCCURS 6 TIMES.
01 INDXX2 PIC S9(1) SYNC USAGE IS COMP.
01 INDXPIC PIC ZZZ9.

01 TERMINAL-MESSAGES.
02 MSG14 PIC X(80) VALUE
' *** ARISCBLC EXECUTION BEGINS ***'.
02 MSG15 PIC X(80) VALUE
' *** ARISCBLC ENDED SUCCESSFULLY ***'.

01 FETCHED-TABLE-HEADERS.
02 MSG16.
03 FILLER PIC X(17) VALUE SPACES.
03 FILLER PIC X(80) VALUE
' *** PRINTOUT OF TABLE INVENTORY UNCHANGED ***'.
02 MSG17.
03 FILLER PIC X(26) VALUE SPACES.
03 FILLER PIC X(80) VALUE
' *** PRINTOUT OF TABLE QUOTATIONS UNCHANGED ***'.
02 MSG18.
03 FILLER PIC X(31) VALUE SPACES.
03 FILLER PIC X(80) VALUE
' *** PRINTOUT OF TABLE SUPPLIERS UNCHANGED ***'.
02 MSG19.

```

    03 FILLER          PIC X(17) VALUE SPACES.
    03 FILLER          PIC X(80) VALUE
'*** PRINTOUT OF TABLE INVENTORY AFTER DELETE ***'.
02 MSG20.
    03 FILLER          PIC X(22) VALUE SPACES.
    03 FILLER          PIC X(80) VALUE
'*** PRINTOUT OF TABLE QUOTATIONS AFTER UPDATE & DELETE ***'.

01 SAMPLE-ORDER-HEADINGS.
02 HEADING1.
    03 FILLER          PIC X(50) VALUE SPACES.
    03 FILLER          PIC X(31) VALUE
'SAMPLE ORDER FOR SAMPLE PROGRAM'.
02 HEADING2.
    03 FILLER          PIC X(50) VALUE SPACES.
    03 FILLER          PIC X(31) VALUE
'
'
02 HEADING3.
    03 FILLER          PIC X(10) VALUE ' NUMBER OF'.
    03 FILLER          PIC X(5) VALUE SPACES.
    03 FILLER          PIC X(11) VALUE 'DESCRIPTION'.
    03 FILLER          PIC X(4) VALUE SPACES.
    03 FILLER          PIC X(8) VALUE 'QUANTITY'.
    03 FILLER          PIC X(5) VALUE SPACES.
    03 FILLER          PIC X(12) VALUE 'COMPANY NAME'.
    03 FILLER          PIC X(4) VALUE SPACES.
    03 FILLER          PIC X(15) VALUE 'COMPANY ADDRESS'.
    03 FILLER          PIC X(22) VALUE SPACES.
    03 FILLER          PIC X(9) VALUE 'PRICE PER'.
    03 FILLER          PIC X(10) VALUE SPACES.
    03 FILLER          PIC X(5) VALUE 'TOTAL'.

02 HEADING4.
    03 FILLER          PIC X(3) VALUE SPACES.
    03 FILLER          PIC X(5) VALUE 'PARTS'.
    03 FILLER          PIC X(22) VALUE SPACES.
    03 FILLER          PIC X(7) VALUE 'ON HAND'.
    03 FILLER          PIC X(62) VALUE SPACES.
    03 FILLER          PIC X(4) VALUE 'UNIT'.
    03 FILLER          PIC X(13) VALUE SPACES.
    03 FILLER          PIC X(4) VALUE 'COST'.

02 HEADING5.
    03 FILLER          PIC X(10) VALUE '_____'.
    03 FILLER          PIC X(5) VALUE SPACES.
    03 FILLER          PIC X(11) VALUE '_____'.
    03 FILLER          PIC X(4) VALUE SPACES.
    03 FILLER          PIC X(8) VALUE '_____'.
    03 FILLER          PIC X(5) VALUE SPACES.
    03 FILLER          PIC X(12) VALUE '_____'.
    03 FILLER          PIC X(4) VALUE SPACES.
    03 FILLER          PIC X(35) VALUE
'
'
    03 FILLER          PIC X(1) VALUE SPACES.
    03 FILLER          PIC X(12) VALUE '_____'.
    03 FILLER          PIC X(4) VALUE SPACES.
    03 FILLER          PIC X(12) VALUE '_____'.

01 INVENTORY-HEADINGS.
02 FILLER             PIC X VALUE SPACES.
02 FILLER             PIC X(23) VALUE SPACES.
02 FILLER             PIC X(6) VALUE 'PARTNO'.
02 FILLER             PIC X(3) VALUE SPACES.
02 FILLER             PIC X(11) VALUE 'DESCRIPTION'.
02 FILLER             PIC X(4) VALUE SPACES.
02 FILLER             PIC X(7) VALUE 'QONHAND'.

```

```

01 INVENTORY-UNDERLINE.
02 FILLER PIC X VALUE SPACES.
02 FILLER PIC X(23) VALUE SPACES.
02 FILLER PIC X(6) VALUE '_____' .
02 FILLER PIC X(3) VALUE SPACES.
02 FILLER PIC X(11) VALUE '_____' .
02 FILLER PIC X(4) VALUE SPACES.
02 FILLER PIC X(7) VALUE '_____' .

01 QUOTATIONS-HEADINGS.
02 FILLER PIC X VALUE SPACES.
02 FILLER PIC X(23) VALUE SPACES.
02 FILLER PIC X(6) VALUE 'SUPPNO' .
02 FILLER PIC X(3) VALUE SPACES.
02 FILLER PIC X(6) VALUE 'PARTNO' .
02 FILLER PIC X(10) VALUE SPACES.
02 FILLER PIC X(5) VALUE 'PRICE' .
02 FILLER PIC X(4) VALUE SPACES.
02 FILLER PIC X(4) VALUE 'TIME' .
02 FILLER PIC X(4) VALUE SPACES.
02 FILLER PIC X(8) VALUE 'QONORDER' .

01 QUOTATIONS-UNDERLINE.
02 FILLER PIC X VALUE SPACES.
02 FILLER PIC X(23) VALUE SPACES.
02 FILLER PIC X(6) VALUE '_____' .
02 FILLER PIC X(3) VALUE SPACES.
02 FILLER PIC X(6) VALUE '_____' .
02 FILLER PIC X(10) VALUE SPACES.
02 FILLER PIC X(5) VALUE '_____' .
02 FILLER PIC X(4) VALUE SPACES.
02 FILLER PIC X(4) VALUE '_____' .
02 FILLER PIC X(4) VALUE SPACES.
02 FILLER PIC X(8) VALUE '_____' .

01 SUPPLIERS-HEADINGS.
02 FILLER PIC X VALUE SPACES.
02 FILLER PIC X(23) VALUE SPACES.
02 FILLER PIC X(6) VALUE 'SUPPNO' .
02 FILLER PIC X(3) VALUE SPACES.
02 FILLER PIC X(4) VALUE 'NAME' .
02 FILLER PIC X(14) VALUE SPACES.
02 FILLER PIC X(7) VALUE 'ADDRESS' .

01 SUPPLIERS-UNDERLINE.
02 FILLER PIC X VALUE SPACES.
02 FILLER PIC X(23) VALUE SPACES.
02 FILLER PIC X(6) VALUE '_____' .
02 FILLER PIC X(3) VALUE SPACES.
02 FILLER PIC X(16) VALUE '_____' .
02 FILLER PIC X(2) VALUE SPACES.
02 FILLER PIC X(34) VALUE
'_____' .

01 SAMPLE-ORDER-STRUC.
02 FILLER PIC X(1) VALUE SPACES.
02 QONORDER-S PIC ZZZZZZZZ9.
02 FILLER PIC X(5) VALUE SPACES.
02 DESCR-S PIC X(10).
02 FILLER PIC X(4) VALUE SPACES.
02 QONHAND-S PIC ZZZZZZZZ9.
02 FILLER PIC X(5) VALUE SPACES.
02 NAME-S PIC X(15).
02 FILLER PIC X(1) VALUE SPACES.
02 ADDR-S PIC X(35).
02 FILLER PIC X(1) VALUE SPACES.

```

```

02 PRICE-S          PIC ZZZZZZZZZ.99.
02 FILLER           PIC X(4) VALUE SPACES.
02 TTL-PRICE-S     PIC ZZZZZZZZZ.99.

01 INVENTORY-STRUC.
02 FILLER           PIC X(24) VALUE SPACES.
02 PARTNO-S        PIC ZZZZZ9.
02 FILLER           PIC X(3) VALUE SPACES.
02 DESCR-S         PIC X(10).
02 FILLER           PIC X(3) VALUE SPACES.
02 QONHAND-S       PIC ZZZZZZZ9.

01 QUOTATIONS-STRUC.
02 FILLER           PIC X(24) VALUE SPACES.
02 SUPPNO-S        PIC ZZZZZ9.
02 FILLER           PIC X(3) VALUE SPACES.
02 PARTNO-S        PIC ZZZZZ9.
02 FILLER           PIC X(3) VALUE SPACES.
02 PRICE-S         PIC ZZZZZZZZZ.99.
02 FILLER           PIC X(3) VALUE SPACES.
02 TIME-S          PIC ZZZZ9.
02 FILLER           PIC X(3) VALUE SPACES.
02 QONORDER-S      PIC ZZZZZZZ9.

01 SUPPLIERS-STRUC.
02 FILLER           PIC X(24) VALUE SPACES.
02 SUPPNO-S        PIC ZZZZZ9.
02 FILLER           PIC X(3) VALUE SPACES.
02 NAME-S          PIC X(15).
02 FILLER           PIC X(3) VALUE SPACES.
02 ADDRVAL-S       PIC X(35).

```

PROCEDURE DIVISION.

OPEN OUTPUT OUT-FILE.

```

* IGNORE SQL WARNINGS AS THEY WILL NOT AFFECT RESULTS
  EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
  EXEC SQL WHENEVER SQLERROR GOTO ERRCHK END-EXEC.
  EXEC SQL WHENEVER NOT FOUND GOTO ERRCHK END-EXEC.

```

DISPLAY MSG14 UPON CONSOLE.

WRITE PRINT-OUT FROM MSG14 AFTER ADVANCING NEWPAGE.

MOVE 'CONNECT ' TO STEP-DENOTER.

EXEC SQL CONNECT :USERID IDENTIFIED BY :PASSW END-EXEC.

```

*****
* PRINT TABLES INVENTORY, QUOTATIONS & SUPPLIERS (UNCHANGED) *
*****

```

WRITE PRINT-OUT FROM MSG16 AFTER ADVANCING NEWPAGE.
PERFORM TABLE1.

WRITE PRINT-OUT FROM MSG17 AFTER ADVANCING NEWPAGE.
PERFORM TABLE2.

WRITE PRINT-OUT FROM MSG18 AFTER ADVANCING NEWPAGE.
PERFORM TABLE3.

```

*****
* FIND MINIMUM PRICE FOR PART NUMBER 310. *
* THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM PRICE *
* FOR ALL OCCURRENCES OF PART NUMBER 310 WITH QONHAND LESS *
* THAN QONORDER OF ZERO. AS PRICE IS A COLUMN IN QUOTATIONS *
* AND QONHAND A COLUMN IN INVENTORY, THE TWO TABLES MUST *
*****

```

```

* BE LINKED VIA A JOIN BETWEEN THE PART NUMBERS IN INVENTORY *
* AND QUOTATIONS. NO CURSOR IS USED BECAUSE THE STATEMENT *
* RETURNS ONLY ONE ROW. *
*****

```

```

MOVE SPACES TO STEP-DENOTER.
MOVE 'SELECT MIN PRICE..., PARTNO=310' TO STEP-DENOTER.

```

```

EXEC SQL SELECT MIN(PRICE)
      INTO :PRICE
      FROM INVENTORY, QUOTATIONS
      WHERE INVENTORY.PARTNO = 310 AND
            QONHAND < 1000 AND
            INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
            QONORDER = 0
      END-EXEC.

```

```

*****
* RETRIEVE DATA OF PART 310 FOR THE SAMPLE ORDER. *
* THE FOLLOWING SELECT STATEMENT RETRIEVES DATA THAT WILL BE *
* USED FOR PRINTING A SAMPLE ORDER FOR PART 310 WITH THE *
* LOWEST PRICE. THE STATEMENT CONTAINS TWO JOINT CONDITIONS *
* BECAUSE DATA FROM ALL THREE TABLES IS REQUIRED. *
*****

```

```

MOVE SPACES TO STEP-DENOTER.
MOVE 'SELECT INVENTORY.PARTNO...,PARTNO=310' TO STEP-DENOTER.

```

```

EXEC SQL SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND,
      PRICE, NAME, ADDRESS, QUOTATIONS.SUPPNO
      INTO :PARTNO, :DESCR, :QONHAND, :PRICE, :NAME,
            :ADDR, :SUPPNO
      FROM INVENTORY, QUOTATIONS, SUPPLIERS
      WHERE INVENTORY.PARTNO = 310 AND
            INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
            PRICE = :PRICE AND
            QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO
      END-EXEC.

```

```

*****
* WRITE OUT HEADINGS FOR SAMPLE ORDER AND PLACE FIRST ORDER *
* OF 1000 PIECES FOR PART 310. *
*****

```

```

WRITE PRINT-OUT FROM HEADING1 AFTER ADVANCING NEWPAGE.
WRITE PRINT-OUT FROM HEADING2 AFTER ADVANCING 0.
WRITE PRINT-OUT FROM HEADING3 AFTER ADVANCING 3.
WRITE PRINT-OUT FROM HEADING4 AFTER ADVANCING 1.
WRITE PRINT-OUT FROM HEADING5 AFTER ADVANCING 0.

```

```

MOVE 1000 TO QONORDER.
COMPUTE TTL-PRICE = PRICE * QONORDER.
PERFORM MOVEOUT.

```

```

*****
* UPDATE QONORDER FOR PART 310 IN TABLE QUOTATIONS. *
* *
*****

```

```

MOVE SPACES TO STEP-DENOTER.
MOVE 'UPDATE QUOTATIONS...,PARTNO=310' TO STEP-DENOTER.

```

```

EXEC SQL UPDATE QUOTATIONS SET QONORDER = 1000
      WHERE PARTNO = :PARTNO AND
            QONORDER = 0 AND

```

```
PRICE = :PRICE AND
SUPPNO = :SUPPNO
END-EXEC.
```

```
*****
*   FIND MINIMUM DELIVERY TIME FOR PART 316                               *
*   THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM DELIVERY*
*   TIME OF ALL OCCURRENCES OF PART NUMBER 316 WITH QONHAND *
*   LESS THAN 700 AND QONORDER ZERO. AS DELIVERY TIME IS A *
*   COLUMN IN QUOTATIONS AND QONHAND A COLUMN IN INVENTORY THE *
*   TWO TABLES MUST BE LINKED VIA A JOIN BETWEEN THE PART *
*   NUMBERS IN QUOTATIONS AND THOSE IN INVENTORY.                 *
*****
```

```
MOVE SPACES TO STEP-DENOTER.
MOVE 'SELECT MIN DELIVERY TIME..PARTNO=316' TO STEP-DENOTER.
```

```
EXEC SQL SELECT MIN(DELIVERY_TIME)
        INTO :TIMEX
        FROM INVENTORY, QUOTATIONS
        WHERE INVENTORY.PARTNO = 316 AND
              QONHAND < 700 AND
              INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
              QONORDER = 0
        END-EXEC.
```

```
*****
*   RETRIEVE DATA OF PART 316 FOR THE SAMPLE ORDER.                   *
*   THE FOLLOWING SELECT STATEMENT RETRIEVES DATA THAT WILL BE *
*   USED FOR PRINTING A SAMPLE ORDER FOR PART 316 WITH THE *
*   LOWEST PRICE. THE STATEMENT CONTAINS TWO JOIN CONDITIONS *
*   BECAUSE DATA FROM ALL THREE TABLES IS REQUIRED.             *
*****
```

```
MOVE SPACES TO STEP-DENOTER.
MOVE 'SELECT INVENTORY.PARTNO.,PARTNO=316' TO STEP-DENOTER.
```

```
EXEC SQL SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND,
        PRICE, NAME, ADDRESS, QUOTATIONS.SUPPNO
        INTO :PARTNO, :DESCR, :QONHAND, :PRICE, :NAME,
        :ADDR, :SUPPNO
        FROM INVENTORY, QUOTATIONS, SUPPLIERS
        WHERE INVENTORY.PARTNO = 316 AND
              INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
              DELIVERY_TIME = :TIMEX AND
              QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO
        END-EXEC.
```

```
*****
*   WRITE AN ORDER FOR PART 316 FOR 700 PIECES.                         *
*   *                                                                     *
*****
MOVE 700 TO QONORDER.
COMPUTE TTL-PRICE = PRICE * QONORDER.
PERFORM MOVEOUT.
```

```
*****
*   UPDATE QONORDER FOR PART 316 IN TABLE QUOTATIONS.                 *
*   *                                                                     *
*****
```

```
MOVE SPACES TO STEP-DENOTER.
MOVE 'UPDATE QUOTATIONS...,PARTNO=316' TO STEP-DENOTER.
```

```
EXEC SQL UPDATE QUOTATIONS SET QONORDER = 700
        WHERE PARTNO = :PARTNO AND
              QONORDER = 0 AND DELIVERY_TIME = :TIMEX AND
```

```

                SUPPNO = :SUPPNO
                END-EXEC.

*****
*   DELETE PART 322 FROM TABLE INVENTORY   *
*****

    MOVE SPACES TO STEP-DENOTER.
    MOVE 'DELETE FROM INVENTORY...,PARTNO=322' TO STEP-DENOTER.

    EXEC SQL DELETE FROM INVENTORY WHERE PARTNO = 322 END-EXEC.

*****
*   DELETE PART 322 FROM TABLE QUOTATIONS *
*****

    MOVE SPACES TO STEP-DENOTER.
    MOVE 'DELETE FROM QUOTATIONS...,PARTNO=322' TO STEP-DENOTER.

    EXEC SQL DELETE FROM QUOTATIONS WHERE PARTNO = 322 END-EXEC.

*****
*   PRINT TABLES INVENTORY, QUOTATIONS & SUPPLIERS (CHANGED) *
*****

    WRITE PRINT-OUT FROM MSG19 AFTER ADVANCING NEWPAGE.
    PERFORM TABLE1.

    WRITE PRINT-OUT FROM MSG20 AFTER ADVANCING NEWPAGE.
    PERFORM TABLE2.

    WRITE PRINT-OUT FROM MSG18 AFTER ADVANCING NEWPAGE.
    PERFORM TABLE3.

    MOVE SPACES TO STEP-DENOTER.
    MOVE 'COMMIT WORK' TO STEP-DENOTER.

    EXEC SQL COMMIT WORK END-EXEC.

    DISPLAY MSG15 UPON CONSOLE.
    WRITE PRINT-OUT FROM MSG15 AFTER ADVANCING NEWPAGE.

    CLOSE OUT-FILE.

    STOP RUN.

*****
*   INTERNAL SUBROUTINE SECTION.           *
*****

    EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

    MOVE SPACES TO STEP-DENOTER.
    MOVE 'SUBROUTINE TABLE1' TO STEP-DENOTER.

TABLE1.
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT PARTNO, DESCRIPTION, QONHAND
        FROM INVENTORY ORDER BY PARTNO
        END-EXEC.
    EXEC SQL OPEN C1 END-EXEC.
        WRITE PRINT-OUT FROM INVENTORY-HEADINGS
            AFTER ADVANCING 3 LINES.
        WRITE PRINT-OUT FROM INVENTORY-UNDERLINE
            AFTER ADVANCING 0.

```

```

        PERFORM FETCH1 UNTIL SQLCODE = 100.
EXEC SQL CLOSE C1 END-EXEC.
FETCH1.
EXEC SQL FETCH C1
        INTO :PARTNO, :DESCR, :QONHAND END-EXEC,
        IF SQLCODE NOT EQUAL 100 THEN
        MOVE PARTNO TO PARTNO-S IN INVENTORY-STRUC,
        MOVE DESCR TO DESCR-S IN INVENTORY-STRUC,
        MOVE QONHAND TO QONHAND-S IN INVENTORY-STRUC,
        WRITE PRINT-OUT FROM INVENTORY-STRUC AFTER ADVANCING 1,
        MOVE ZEROES TO PARTNO,
        MOVE SPACES TO DESCR,
        MOVE SPACES TO INVENTORY-STRUC.

MOVE SPACES TO STEP-DENOTER.
MOVE 'SUBROUTINE TABLE2' TO STEP-DENOTER.

TABLE2.
EXEC SQL DECLARE C2 CURSOR FOR SELECT SUPPNO, PARTNO, PRICE,
        DELIVERY TIME, QONORDER
        FROM QUOTATIONS ORDER BY
        SUPPNO, PARTNO END-EXEC.
EXEC SQL OPEN C2 END-EXEC.
        WRITE PRINT-OUT FROM QUOTATIONS-HEADINGS
                AFTER ADVANCING 3 LINES.
        WRITE PRINT-OUT FROM QUOTATIONS-UNDERLINE
                AFTER ADVANCING 0.
        PERFORM FETCH2 UNTIL SQLCODE = 100.
EXEC SQL CLOSE C2 END-EXEC.
FETCH2.
EXEC SQL FETCH C2
        INTO :SUPPNO, :PARTNO,
        :PRICE, :TIMEX, :QONORDER END-EXEC,
        IF SQLCODE NOT EQUAL 100 THEN,
        MOVE SUPPNO TO SUPPNO-S IN QUOTATIONS-STRUC,
        MOVE PARTNO TO PARTNO-S IN QUOTATIONS-STRUC,
        MOVE PRICE TO PRICE-S IN QUOTATIONS-STRUC,
        MOVE TIMEX TO TIME-S IN QUOTATIONS-STRUC,
        MOVE QONORDER TO QONORDER-S IN QUOTATIONS-STRUC,
        WRITE PRINT-OUT FROM QUOTATIONS-STRUC AFTER ADVANCING 1,
        MOVE ZEROES TO SUPPNO,
        MOVE ZEROES TO PARTNO,
        MOVE ZEROES TO PRICE,
        MOVE ZEROES TO TIMEX,
        MOVE ZEROES TO QONORDER,
        MOVE SPACES TO QUOTATIONS-STRUC.

MOVE SPACES TO STEP-DENOTER.
MOVE 'SUBROUTINE TABLE3' TO STEP-DENOTER.

TABLE3.
EXEC SQL DECLARE C3 CURSOR FOR SELECT SUPPNO, NAME,
        ADDRESS FROM SUPPLIERS
        ORDER BY SUPPNO END-EXEC.
EXEC SQL OPEN C3 END-EXEC.
        WRITE PRINT-OUT FROM SUPPLIERS-HEADINGS
                AFTER ADVANCING 3 LINES.
        WRITE PRINT-OUT FROM SUPPLIERS-UNDERLINE
                AFTER ADVANCING 0.
        PERFORM FETCH3 UNTIL SQLCODE = 100.
EXEC SQL CLOSE C3 END-EXEC.
FETCH3.
EXEC SQL FETCH C3
        INTO :SUPPNO, :NAME, :ADDR END-EXEC,
        IF SQLCODE NOT EQUAL 100 THEN,
        MOVE SUPPNO TO SUPPNO-S IN SUPPLIERS-STRUC,
        MOVE NAME TO NAME-S IN SUPPLIERS-STRUC,

```



```

MOVE ADDRVAL TO ADDRVAL-S IN SUPPLIERS-STRUC,
WRITE PRINT-OUT FROM SUPPLIERS-STRUC AFTER ADVANCING 1,
MOVE SPACES TO SUPPLIERS-STRUC.
MOVE SPACES TO ADDRVAL.

```

```
MOVEOUT.
```

```

MOVE QONORDER TO QONORDER-S IN SAMPLE-ORDER-STRUC.
MOVE DESCR TO DESCR-S IN SAMPLE-ORDER-STRUC.
MOVE QONHAND TO QONHAND-S IN SAMPLE-ORDER-STRUC.
MOVE NAME TO NAME-S IN SAMPLE-ORDER-STRUC.
MOVE ADDRVAL TO ADDR-S IN SAMPLE-ORDER-STRUC.
MOVE PRICE TO PRICE-S IN SAMPLE-ORDER-STRUC.
MOVE TTL-PRICE TO TTL-PRICE-S IN SAMPLE-ORDER-STRUC.
WRITE PRINT-OUT FROM SAMPLE-ORDER-STRUC AFTER ADVANCING 2.
MOVE ZEROES TO QONORDER-S IN SAMPLE-ORDER-STRUC.
MOVE SPACES TO DESCR-S IN SAMPLE-ORDER-STRUC.
MOVE ZEROES TO QONHAND-S IN SAMPLE-ORDER-STRUC.
MOVE SPACES TO NAME-S IN SAMPLE-ORDER-STRUC.
MOVE SPACES TO ADDR-S IN SAMPLE-ORDER-STRUC.
MOVE ZEROES TO PRICE-S IN SAMPLE-ORDER-STRUC.
MOVE SPACES TO PRINT-OUT.

```

```
ERRCHK.
```

```

*****
* THE FOLLOWING ROUTINE PRINTS THE SQLCA STRUCTURE:
*
* - SQLCODE = SQL RETURN CODE
* - SQLERRM = SQL ERROR MESSAGE
* - SQLERRP = MODULE DETECTING ERROR
* - SQLERRD = INTERNAL ERROR VALUES
* - SQLWARN = SQL WARNING STRUCTURE
*
*****
DISPLAY '*****' UPON CONSOLE.
DISPLAY '* PROGRAM ERROR ROUTINE ENTERED *' UPON CONSOLE.
DISPLAY '* CHECK SYSPRINT FOR ERROR CODES*' UPON CONSOLE.
DISPLAY '* CHANGES WILL BE BACKED OUT *' UPON CONSOLE.
DISPLAY '*****' UPON CONSOLE.
MOVE SQLCODE TO DECODED-SQLCODE.
DISPLAY 'PROGRAM ERROR ROUTINE ENTERED'.
DISPLAY '*****'.
DISPLAY 'A PROBLEM HAS BEEN DETECTED IN THE '.
DISPLAY STEP-DENOTER.
DISPLAY 'THE FOLLOWING ERROR CODES SHOULD AID YOU IN'.
DISPLAY 'PROBLEM DETERMINATION OF THE SQL STATEMENT.'.
DISPLAY '*****'.
DISPLAY 'SQLCODE : ' DECODED-SQLCODE.
DISPLAY 'SQLERRM : ' SQLERRMC.
DISPLAY 'SQLERRP : ' SQLERRP.
PERFORM ERRD VARYING INDX2 FROM 1 BY 1 UNTIL INDX2 = 7.
IF SQLWARNO NOT EQUAL 'W'
    THEN GO TO BACKOUT,
    ELSE DISPLAY 'SQLWARNO: ' SQLWARNO,
        DISPLAY 'SQLWARN1: ' SQLWARN1,
        DISPLAY 'SQLWARN2: ' SQLWARN2,
        DISPLAY 'SQLWARN3: ' SQLWARN3,
        DISPLAY 'SQLWARN4: ' SQLWARN4,
        DISPLAY 'SQLWARN5: ' SQLWARN5,
        DISPLAY 'SQLWARN6: ' SQLWARN6,
        DISPLAY 'SQLWARN7: ' SQLWARN7,
        DISPLAY 'SQLWARN8: ' SQLWARN8,
        DISPLAY 'SQLWARN9: ' SQLWARN9,
        DISPLAY 'SQLWARNA: ' SQLWARNA,
        GO TO BACKOUT.
ERRD. MOVE SQLERRD (INDX2) TO DECODED-SQLERRD (INDX2).
* MOVE INDX2 TO INDXPIC.

```

```
        DISPLAY 'SQLERRD', INDX2, ': ', DECODED-SQLERRD (INDX2).
BACKOUT.
*****
* 'WHENEVER' RESET TO 'CONTINUE' IN THE EVENT THAT THE ROLLBACK *
* WORK STATEMENT FAILS TO AVOID LOOP IN ERROR ROUTINE.          *
*****

MOVE SPACES TO STEP-DENOTER.
MOVE 'SUBROUTINE BACKOUT' TO STEP-DENOTER.

EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
EXEC SQL ROLLBACK WORK END-EXEC.
      STOP RUN.
```

Rules for Using SQL in COBOL

This section lists, for your reference, all the rules for embedding SQL statements within a COBOL program.

Placement and Continuation of SQL Statements

All SQL statements must be placed in columns 12 to 72. Place all non-declarative SQL statements (including all cursor-related statements) in the Procedure Division.

The rules for continuation of SQL keywords from one line to the next are the same as the COBOL rules for the continuation of words and constants. If a string-constant is continued from one line to the next, the first non-blank character in that next line must be an apostrophe ('). If a delimited SQL identifier (such as "EMP TABLE") is continued from one line to the next, the first non-blank character in that next line must be a quote mark ("). COBOL comment lines, identified by an * in column 7, can be coded within an embedded statement.

Delimiting SQL Statements

Delimiters are required on all SQL statements to help SQL/DS distinguish them from regular COBOL statements. You must precede each SQL statement with "EXEC SQL" and terminate each one with "END-EXEC". Any desired COBOL punctuation, such as a period, can be placed after the "END-EXEC". (No punctuation is required after the "END-EXEC" that terminates an SQL statement.) For example, suppose an SQL statement occurs as one of several statements nested inside a COBOL IF-statement. In this instance, the SQL statement should not be followed by a period.

"EXEC SQL" must be specified within one line; the same is true for "END-EXEC".

If an SQL statement appears within an IF sentence such that a COBOL ELSE clause immediately follows the SQL statement, the COBOL ELSE clause must begin with the word "ELSE". In addition, this "ELSE" must be contained entirely on one line. (No continuation is allowed for the word "ELSE".)

SQL WHENEVER and DECLARE CURSOR statements should *not* be the only contents of COBOL IF or ELSE clauses. The SQL/DS preprocessor does not generate COBOL code for these statements.

If an SQL statement is to terminate a COBOL IF sentence, a period should immediately follow "END-EXEC" with no intervening blanks. A blank should follow the period.

Because a COBOL statement can be immediately preceded by a paragraph name, an embedded statement can also be immediately preceded by a paragraph name. Similarly, an embedded statement in the Procedure Division can be immediately followed by a separator period.

Declaring Host Variables

You must declare all host variables to be used in SQL statements. The declarations, as shown in the earlier example program, must appear in an SQL declare section that begins with

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```

and ends with

```
EXEC SQL END DECLARE SECTION END-EXEC.
```

The declarations in the SQL declare section follow normal COBOL continuation rules. The declare sections can be located in the Working-Storage Section, the File Section, or the Linkage Section of the Data Division. You can have more than one SQL declare section within a program.

Place within these SQL declare sections the data description entries for all the host variables. You can use the variables appearing in these SQL declare sections in regular COBOL statements as well as in SQL statements.

You can also place data description entries for non-host variables in the SQL declare section. The COBOL preprocessor scans the SQL declare section(s) for potential host variable declarations and performs error-checking on only those data description entries that may be host variable declarations. All other data description entries within the SQL declare section(s) are ignored. Thus, it is possible, but not recommended, to place all data description entries within an SQL declare section.

Cursor declarations should appear in the Procedure Division.

The rules for declaring variables within SQL declare sections are as follows:

- Host variables can be named using COBOL rules, but they are limited to 18 characters. The SQL/DS preprocessor changes hyphens (-) in host variable names in SQL statements to underscores (_), conforming to SQL/DS naming rules. Note that GO TO labels in SQL/DS WHENEVER statements are also restricted to 18 characters.
- Variables named in the SQL declare sections must have data descriptions like those in Figure 42 on page 417. Single-level variables can utilize either level 01 or level 77. The levels for varying-length strings are described below. Other valid COBOL clauses such as SYNC or VALUE can be added to those given in Figure 42. The OCCURS, SIGN, JUSTIFIED, and BLANK WHEN ZERO clauses can not be used. Any data description entry except the LENGTH part of a varying-length character string may be followed by one or more REDEFINES or RENAMES entries (however, the names in these entries cannot be used in SQL statements). Level-88 entries (“condition-name” entries associated with another variable) are permitted in the SQL declare sections. Entries with the name “FILLER” are ignored by the preprocessor.
- Variables corresponding to varying-length strings in SQL/DS must have group level 01 and the group must contain two level-49 elementary items, as shown in Figure 42. The group, and the two elementary items, can have any desired

names. The first elementary item must have PICTURE S9(n), where *n* is an integer from 1-4, and is used to represent the length of the string. The second elementary item must have PICTURE X(n) where *n* is the maximum length of the string, and is used to contain the value of the string.

The variable should be referred to in SQL statements by its group-name only. If the variable is being used for storing a varying-length string in the data base, SQL/DS will store only as many characters as indicated by the length-item. If SQL/DS is retrieving a string from the data base into the variable, it will indicate in the length-item the number of characters which were retrieved into the value-item.

- You should not give any variable a name beginning with SQL or RDI, because these names are reserved for SQL/DS use.

Using Host Variables in SQL Statements

When you place host variables within an SQL statement, you must precede each such variable with a colon (:). The colon distinguishes the host variables from the SQL identifiers (such as PARTNO). When the same variable is used outside of an SQL statement, do not use a colon.

A host variable can represent a data value, but not an SQL identifier. For example, you cannot assign a character constant, such as 'MUSICIANS', to a host variable, and then use that host variable in a CREATE TABLE statement to represent the table name. This pseudo-code sequence is invalid:

```
TT = 'MUSICIANS'  
CREATE TABLE :TT (NAME ...
```

Incorrect

When performing subtraction in an SQL statement, delimit the minus sign (-) with blanks:

```
blanks  
 | |  
 V V  
QUANT - :ORDER-AMOUNT
```

QUOTE Parameter of the SQL/DS Preprocessor

If the COBOL compiler QUOTE option is used, the QUOTE (or Q) option of the SQL/DS preprocessor should also be specified. You should use a single quote (') to delineate constants used in embedded SQL statements, regardless of the COBOL compiler QUOTE option.

COPYBOOKS

You should not use COPYBOOKS when an SQL host variable is involved.

Using the INCLUDE Command

To include the external secondary input, specify:

```
EXEC SQL INCLUDE text-name END-EXEC.
```

at the point in the source code where the secondary input is to be included. Text-name is the filename of a CMS file with a “COBCOPY” filetype and located on a CMS minidisk accessed by the user.

The INCLUDE command can appear anywhere within the File, Linkage, or Working Storage Sections of the Data Division and anywhere within the Procedure Division, including the Declaratives Section, if one is used. Note that the INCLUDE command is the only type of SQL statement that is allowed within the Declaratives Section of a Procedure Division.

COBOL Data Conversion Notes

COBOL variables used in SQL statements must be type-compatible with the columns of the tables with which they are to be used (stored, retrieved, or compared).

A column of type INTEGER, SMALLINT, DECIMAL, or FLOAT is compatible with a COBOL variable of PICTURE S9(4) COMPUTATIONAL, PICTURE S9(9) COMPUTATIONAL, PICTURE S9(p)[V9(q)] COMPUTATIONAL-3, or COMPUTATIONAL-2. Of course, an overflow condition may occur if, for example, an INTEGER data item is retrieved into a PICTURE S9(4) variable, and its current value is too large to fit.

The three types of character data (fixed-length, varying-length, and LONG VARCHAR) and three types of DBCS data (fixed-length, varying-length, and LONG VARGRAPHIC) are also considered compatible. SQL/DS automatically converts a varying-length string to a fixed-length string, and vice-versa, when necessary. If a varying-length string is converted to a fixed-length string, it is truncated or padded on the right with blanks to the correct length. SQL/DS also truncates or pads with blanks if a fixed-length string is assigned to another fixed-length string of a different size (for example, a variable of PICTURE X(12) is stored in a column of type CHAR(18)).

Refer to “Data Conversion” on page 76 for a data conversion summary.

COBOL Comments

The SQL/DS preprocessor scans past COBOL NOTE-type comments and line comments defined via an '*' in column 7. Line comments identified by a '/' in column 7 are not recognized by the preprocessor.

SQL Statements in COBOL Subprograms

There is considerable SQL initialization processing in connection with invoking a program that contains SQL statements. The performance of your program may be considerably reduced by locating SQL statements in called subprograms, rather than in the main program. The initialization processing by SQL/DS is approximately proportional to the number of SQL statements involved. Your decision to locate SQL statements in subprograms will depend on the number of SQL statements, the frequency of subprogram invocation, and other application-dependent considerations.

Using the Double-Byte Character Set (DBCS)

If your program contains DBCS data representations, the following sequence of processing is necessary:

- SQL/DS COBOL Preprocessor
- COBOL Kanji Preprocessor
- CICS/DOS/VS Translator, if necessary
- COBOL Compiler.

The COBOL Kanji preprocessor is a PRPQ that comes in two versions. The one for the VM/SP environment is OS/VS Utility Program -- Kanji, 5799-BBA (RPQ reference number 7F0095).

The DBCS constant in SQL statements embedded in COBOL programs has the following format:

```
G'so...si'
```

That is, SQL DBCS constants are used.

Since they are not within the so/si delimiters, the letter G and the apostrophes (') are single-byte, EBCDIC characters, X'C7' and X'7D' respectively. (The ellipsis, ..., represents a DBCS string.) The left byte of a DBCS byte-pair must not be X'0F', since this would signal exit from DBCS encoding. The characters contained within the so/si delimiters can be of any bit configuration, but no mixed data (DBCS and EBCDIC) is allowed. That is, there must be an even number of bytes between the so and the si delimiters. In this form of the constant, the DBCS character for apostrophe (X'427D') is not doubled to obtain a single DBCS apostrophe character, as opposed to DBCS constants used in PL/I programs.

The SQL/DS COBOL preprocessor does not support options for changing the encoding for the so/si characters. They must be defined as X'0E' and X'0F'.

When the DBCS option is set to YES, SQL identifiers with DBCS characters can be used in SQL statements. For more information on SQL identifiers, see “General Rules for Naming Data Objects” on page 74.

When the DBCS option is set to YES, both character string constants in SQL statements and character string constants in COBOL statements can contain DBCS strings which are enclosed by so and si. However, no DBCS string can span across a line in the program. An apostrophe (X'7D') in a DBCS string does not terminate a character string constant and does not have to be duplicated. Therefore, note that if there is a X'7D' between so and si in a character string constant in a COBOL statement, it cannot be correctly compiled by the COBOL compiler.

SQL Error Handling

The SQLCA return code structure that is required for SQL/DS can be declared in two ways:

1. You may write:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

in the Working-Storage Section of your source program. The SQL/DS preprocessor replaces this with a declaration of the SQLCA structure.

2. You may declare the SQLCA yourself in the Working-Storage Section as shown in Figure 41.


```

01 SQLCA.
   05 SQLCAID      PIC X(8) .
   05 SQLCABC      S9(9) COMPUTATIONAL.
   05 SQLCODE      PIC S9(9) COMPUTATIONAL.
   05 SQLERM.
      49 SQLERRML  PIC S9(4) COMPUTATIONAL.
      49 SQLERRMC  PIC X(70) .
   05 SQLERP      PIC X(8) .
   05 SQLERRD      OCCURS 6 TIMES
                   PIC S9(9) COMPUTATIONAL.

   05 SQLWARN.
      10 SQLWARN0  PIC X(1) .
      10 SQLWARN1  PIC X(1) .
      10 SQLWARN2  PIC X(1) .
      10 SQLWARN3  PIC X(1) .
      10 SQLWARN4  PIC X(1) .
      10 SQLWARN5  PIC X(1) .
      10 SQLWARN6  PIC X(1) .
      10 SQLWARN7  PIC X(1) .
      10 SQLWARN8  PIC X(1) .
      10 SQLWARN9  PIC X(1) .
      10 SQLWARNA  PIC X(1) .
   05 SQLEXT      PIC X(5) .

```

Figure 41. SQLCA Structure (in COBOL)

A COBOL program containing SQL/DS statements must have a Working-Storage Section. The meanings of the fields within the SQLCA are discussed under “Error Handling” on page 202.

In COBOL, the object of a GO TO in the SQL WHENEVER statement must be a section-name or an unqualified paragraph-name.

Dynamic SQL Statements in COBOL

The COBOL preprocessor does not support the DESCRIBE statement, and supports only Format 1 of the EXECUTE, OPEN, and FETCH statements. The SQLDA structure does not apply.

For COBOL, the string-spec in PREPARE and EXECUTE IMMEDIATE must be in the same format as the SQL VARCHAR data type (you must set the proper length) or a quoted string. If a quoted string is used, its length is limited to 120 characters (the maximum length allowed for COBOL constants). In addition, you cannot use a single (') or double (") quote within a COBOL constant that is the object of a PREPARE or EXECUTE IMMEDIATE.

Data Types

Description	SQL/DS Keyword	Equivalent COBOL Declaration
A binary integer of 31 bits, plus sign.	INTEGER	01 PICTURE S9(9) COMPUTATIONAL.
A binary integer of 15 bits, plus sign.	SMALLINT	01 PICTURE S9(4) COMPUTATIONAL.
A packed decimal number, precision m, scale n ($1 \leq m \leq 15$ and $0 \leq n \leq m$). In storage the number occupies an even number of bytes up to a maximum of 8 bytes. Precision is the total number of digits. Scale is the number of those digits that are to the right of the decimal point.	DECIMAL(m[,n])	01 PICTURE S9(p)[V9(q)] COMPUTATIONAL-3. Where $p + q = m$ and $q = n$
A double-precision (8-byte) floating point number, in standard System/370 floating point format.	FLOAT	COMPUTATIONAL-2.
A fixed-length character string of length n where $n \leq 254$.	CHAR(n)	01 S PICTURE X(n).
A varying-length character string of maximum length n, where $n \leq 254$. (Only the actual length is stored in the data base.)	VARCHAR(n)	01 S. 49 S-LENGTH PICTURE S9(4) COMPUTATIONAL. 49 S-VALUE PICTURE X(n).
A varying-length character string of maximum length 32767 bytes, subject to certain usage limitations.	LONG VARCHAR	01 S. 49 S-LENGTH PICTURE S9(4) COMPUTATIONAL. 49 S-VALUE PICTURE X(n).
A fixed-length DBCS character string of n DBCS characters where $n \leq 127$.	GRAPHIC(n)	01 GNAME PICTURE G(n) USAGE DISPLAY-1.
A varying-length DBCS character string of n DBCS characters where $n \leq 127$.	VARGRAPHIC(n)	01 GNAME. 49 GGLEN PICTURE S9(4) COMPUTATIONAL. 49 GGVAL PICTURE G(n). USAGE DISPLAY-1.

Figure 42 (Part 1 of 2). SQL/DS Data Types for COBOL

Description	SQL/DS Keyword	Equivalent COBOL Declaration
A varying-length DBCS character string of maximum length 16383, subject to certain usage limitations.	LONG VARGRAPHIC	01 XNAME. 49 XNAMLEN PICTURE S9(4) COMPUTATIONAL. 49 XNAMVAL PICTURE G(n). USAGE DISPLAY-1.

Figure 42 (Part 2 of 2). SQL/DS Data Types for COBOL

Notes:

1. Level Number 77 (but not other levels) can be used in place of Level Number 01 for all but VARCHAR and VARGRAPHIC data types.
2. "USAGE" or "USAGE IS" is optional before "COMPUTATIONAL" and "DISPLAY-1".
3. "COMPUTATIONAL" can be abbreviated "COMP". "PICTURE" can be abbreviated "PIC".
4. INTEGER and SMALLINT data types can have sliding ranges. For example, if you wish to declare a SMALLINT variable that you know will remain very small, you could use S9(2) instead of S9(4). Or, you could declare an integer with a range of S9(7) instead of S9(9). However, only the ranges shown in the above table allow for the largest possible values of SMALLINT and INTEGER. Truncation may occur if you elect to declare smaller ranges.
5. For COMPUTATIONAL types, 9s may be repeated rather than using the repetition factors in parentheses (that is, 9999 instead of 9(4)). The same is true for the Xs in the character types and Gs in the DBCS character types.
6. The word "IS" can follow "PICTURE" or "PIC" to improve readability.
7. In DECIMAL data types, precision is the total number of digits. Scale is the number of digits to the right of the decimal point. If an even precision is specified, SQL/DS assumes the next higher (odd) precision.
8. When a VALUE clause is used for host variables of the form "PIC S9(4) COMP", 9999 is the highest value accepted by COBOL. If you specify the COBOL NOTRUNC option, however, a value up the 32767 can be *moved* into the host variable. If host variables are to contain LONG VARCHAR or LONG VARGRAPHIC data where the length exceeds 9999, the NOTRUNC option must be set.

Additional COBOL Program Example

The following program is another example of how to embed SQL statements in a COBOL program.

Sometimes the BETWEEN function causes performance problems because it does not have very good information on path selection. This program shows an alternate way of performing the BETWEEN function, when these performance problems exist.

```
*****
* THIS IS A SAMPLE COBOL PROGRAM THAT PERFORMS THE      *
* BETWEEN FUNCTION USING DYNAMIC SQL STATEMENTS          *
*                                                         *
* WHEN THE USER CODES A STATIC SQL STATEMENT USING     *
* HOST VARIABLES FOR THE BETWEEN FUNCTION, THE SQL/DS   *
* OPTIMIZER DOES NOT HAVE VERY GOOD INFORMATION FOR    *
* PATH SELECTION. THEREFORE, THE USER SHOULD USE      *
* SQL DYNAMIC STATEMENTS TO PERFORM THIS FUNCTION.     *
*****
IDENTIFICATION DIVISION.
PROGRAM-ID. BETWDYNC.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    C01 IS NEWPAGE.

INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OUT-FILE ASSIGN TO SYS008-UR-1403-S-OUT-FILE.
DATA DIVISION.
FILE SECTION.
FD  OUT-FILE
   LABEL RECORDS ARE OMITTED
   DATA RECORD IS PRINT-OUT.
01  PRINT-OUT      PIC X(130).
WORKING-STORAGE SECTION.
*****
*  HOST VARIABLE DECLARATION SECTION                      *
*****
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
77  PARTNO        PIC S9(4) COMP.
77  QONHAND       PIC S9(9) COMP.
01  DESCR.
   49  DESCR-LEN  PIC S9(4) COMP.
   49  DESCR-DATA PIC X(24).
01  DESCR-IND PIC S9(4) COMP.
01  USERID      PIC X(8) VALUE 'SQLDBA '.
01  PASSW       PIC X(8) VALUE 'SQLDBAPW'.
01  ESTRING.
   49  E-LEN      PIC S9(4) COMP VALUE +66.
   49  P1         PIC X(47) VALUE
       'SELECT * FROM INVENTORY WHERE QONHAND BETWEEN '.
   49  P2         PIC 9999999.
   49  P3         PIC X(5) VALUE ' AND '.
   49  P4         PIC 9999999.
EXEC SQL END DECLARE SECTION END-EXEC.
EXEC SQL INCLUDE SQLCA END-EXEC.
01  DECODED-SQLCODE PIC -----999.
01  ARRAY-SQLERRD.
   02  DECODED-SQLERRD PIC -----999 OCCURS 6 TIMES.
```

```

01  INDX2                PIC S9(1) SYNC USAGE IS COMP.
01  R15                  PIC -9999.
01  INVENTORY-HEADINGS.
    02  FILLER           PIC X VALUE SPACES.
    02  FILLER           PIC X(23) VALUE SPACES.
    02  FILLER           PIC X(6) VALUE 'PARTNO'.
    02  FILLER           PIC X(3) VALUE SPACES.
    02  FILLER           PIC X(11) VALUE 'DESCRIPTION'.
    02  FILLER           PIC X(17) VALUE SPACES.
    02  FILLER           PIC X(7) VALUE 'QONHAND'.
01  INVENTORY-UNDERLINE.
    02  FILLER           PIC X VALUE SPACES.
    02  FILLER           PIC X(23) VALUE SPACES.
    02  FILLER           PIC X(6) VALUE '_____' .
    02  FILLER           PIC X(3) VALUE SPACES.
    02  FILLER           PIC X(11) VALUE '_____' .
    02  FILLER           PIC X(17) VALUE SPACES.
    02  FILLER           PIC X(7) VALUE '_____' .
01  INVENTORY-STRUC.
    02  FILLER           PIC X(24) VALUE SPACES.
    02  PARTNO-S         PIC ZZZZZ9.
    02  FILLER           PIC X(3) VALUE SPACES.
    02  DESCR-S         PIC X(24).
    02  FILLER           PIC X(2) VALUE SPACES.
    02  QONHAND-S       PIC ZZZZZZZZ9.
01  END-MSG.
    02  FILLER           PIC X(20) VALUE SPACES.
    02  FILLER           PIC X(17) VALUE 'END OF ANSWER SET'.
PROCEDURE DIVISION.
    OPEN OUTPUT OUT-FILE.
    EXEC SQL WHENEVER SQLERROR GOTO ERRCHK END-EXEC.
    EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
    EXEC SQL WHENEVER NOT FOUND GOTO END-DATA END-EXEC.
    EXEC SQL CONNECT :USERID IDENTIFIED BY :PASSW END-EXEC.
    MOVE RETURN-CODE TO R15.
    GET-RANGE-VALUES.
    *****
    * THERE ARE MANY WAYS TO GET THE RANGE VALUES FOR      *
    * THE BETWEEN FUNCTION INTO YOUR PROGRAM. THIS          *
    * PROGRAM WILL SIMPLE MOVE VALUES INTO THE DATA      *
    * STRING OF THE SQL COMMAND.                            *
    *****
    MOVE 75 TO P2.
    MOVE 100 TO P4.
    PREPARE-REQUEST.
    EXEC SQL PREPARE STAT1 FROM :ESTRING END-EXEC.
    DECLARE-CURSOR.
    EXEC SQL DECLARE C1 CURSOR FOR STAT1 END-EXEC.
    OPEN-CURSOR.
    EXEC SQL OPEN C1 END-EXEC.
    MOVE RETURN-CODE TO R15.
    WRITE PRINT-OUT FROM INVENTORY-HEADINGS
        AFTER ADVANCING NEWPAGE.
    WRITE PRINT-OUT FROM INVENTORY-UNDERLINE
        AFTER ADVANCING 0.
    GET-NEXT.
    MOVE SPACES TO DESCR-DATA.
    EXEC SQL FETCH C1 INTO :PARTNO, :DESCR:DESCR-IND, :QONHAND
        END-EXEC.
    MOVE PARTNO TO PARTNO-S IN INVENTORY-STRUC.
    IF DESCR-IND < 0 THEN MOVE 'NULL DATA' TO DESCR-S
        IN INVENTORY-STRUC
    ELSE IF DESCR-LEN = 0 THEN MOVE 'ZERO LENGTH' TO DESCR-S
        IN INVENTORY-STRUC
    ELSE MOVE DESCR-DATA TO DESCR-S IN INVENTORY-STRUC.
    MOVE QONHAND TO QONHAND-S IN INVENTORY-STRUC.

```


Appendix E. Assembler Considerations

This appendix contains a sample programs that illustrates the use of SQL within Assembler language. Following the sample are specific rules for using SQL in Assembler language.

Acquiring the SQLDSECT Area

The assembler preprocessor puts all the variables and structures it generates within a DSECT named SQLDSECT. The preprocessor also generates a variable, called SQLDSIZ, that contains the length of the SQLDSECT DSECT in bytes. Thus, for all Assembler programs, you must provide an area of size SQLDSIZ, zero the area, and provide addressability to the SQLDSECT DSECT.

You must use the CMS DMSFREE macro GETMAIN (CMS OS/VS program) or GETVIS (CMS VSE program) to acquire storage. Note that SQLDSIZ is in bytes and that you need the length in doublewords. Figure 43 shows sample pseudo-code that can be used to acquire the SQLDSECT area.

```
TESTNAME CSECT
STM 14,12,12(13)
BALR regx,0
USING *,regx
LA regy,7(0,0)
A regy,SQLDSIZ
SRL regy,3
(save computed doubleword length for DMSFRET)
LR 0,regy
DMSFREE DWORDS=(0)
LR regz,1
USING SQLDSECT,regz
(add code to zero the area)
.
.
.
(add code to free storage via DMSFRET)
END
```

This area is needed only until the program is finished executing all SQL statements, at which time the area should be freed (DMSFRET).

Figure 43. Acquiring the SQLDSECT Area for VM/SP Applications

If you know the approximate size of the SQLDSECT that will be generated in your program, you can define an area (AREA DS CLxxxx) within your program and use this as your SQLDSECT area. Your program will not be re-entrant if you use this method.

The preprocessor generates the code to calculate SQLDSIZ directly in front of the last statement in the source program. It is recommended that the last statement be an END statement.

If the Assembler preprocessor is run with the CHECK option, SQLDSECT and SQLDSIZ will not be generated. Errors will occur if you attempt to assemble the output generated by the preprocessor when the CHECK option is specified. See "Preprocessing and Running the Program" on page 183 in Chapter 2 for more information about preprocessor parameters.

Note that you must provide a save area for all Assembler programs.

Performance Considerations for the SQLDSECT Area

There are two performance considerations about the SQLDSECT area that you should be aware of:

1. Acquire and clear the SQLDSECT area only once.

The example shown in Figure 43 assumes that the TESTNAME is entered once. If TESTNAME is a subroutine of a mainline module, and if TESTNAME is invoked many times, you should acquire the SQLDSECT in the mainline module. The following is an example of how this may be done:

- a. In TESTNAME add an entry card as follows:

```
ENTRY SQLDSIZ
```

This allows the field containing the size information for the SQLDSECT area to be accessed externally.

- b. The mainline module can now access the size information using the following sequence:

```
L regy,=V(SQLDSIZ)  GET POINTER TO FIELD CONTAINING SIZE
L 0,0(,regy)        SET LENGTH
GETVIS ADDRESS=(1),LENGTH=(0)
LR regy,1           SAVE POINTER TO SQLDSECT
(Zero the SQLDSECT area.)
```

- c. When the mainline module calls TESTNAME, it should pass the pointer to the SQLDSECT. Assuming that regy still contains the pointer, TESTNAME simply issues the appropriate USING as follows:

```

TESTNAME CSECT
        STM    14,12,12(13)
        BALR   regx,0
        USING  SQLDSECT,regy
        .
        .

```

Depending on how many times TESTNAME is invoked, the above could be an important performance consideration. Using the technique reduces the path length because you only need to get, clear, and free storage once. Further, the cleared SQLSECT area serves as a “first pass” flag for the SQL/DS batch/ICCF and CMS resource managers. Thus, by letting the mainline module initialize the SQLSECT area only once, you further avoid significant SQL/DS resource manager “first pass” processing.

2. Provide only one SQLDSECT area.

If you structure an application so that the mainline module invokes several modules that each contain SQL commands, you need to provide only one SQLDSECT area. The area that you provide must be the largest SQLDSECT area. For example, suppose the mainline module invokes MODA and MODB, each of which contain SQL commands. MODA and MODB have different SQLDSECT area requirements. The mainline module must satisfy the larger of the two requirements.

By inserting the following into MODA and MODB you could allow the mainline module to calculate the SQLDSECT area requirement:

```

INTO MODA:                                INTO MODB:
MODADSIZ DC A(SQLDSIZ)                    MODBDSIZ DC A(SQLDSIZ)
        ENTRY MODADSIZ                      ENTRY MODBDSIZ
        .                                    .
        .                                    .

```

The mainline module could reference the above entries and provide for the maximum SQLDSECT area. The following shows how, for example, the mainline module could determine the requirement of MODA:

```

L   regy,=V(MODADSIZ)  GET POINTER TO POINTER FIELD
L   regy,0(,regy)      GET POINTER TO FIELD CONTAINING SIZE
L   0,0(,regy)         SET LENGTH.

```

The same technique could be used to access the SQLDSIZ of MODB. Given the two SQLDSIZ values, the mainline module should provide for a SQLDSECT area equal in size to the greater SQLDSIZ value.

By using only one SQLDSECT area for your application, you reduce the storage requirement and minimize the “first pass” processing.

ARISASMC -- Assembler Sample Program

ARISASMC is an Assembler language sample program for VM/SP systems. The source code of these programs begins on the next page. You can learn most of the rules for using SQL within Assembler language just by scanning through these programs. Note, in particular, how the programs satisfy the requirements of the application prolog and epilog. Near the beginning of the programs, all the host variables are declared, the SQLDSECT area is acquired (and set to zero), error handling is defined, and a connection is established with SQL/DS. Near the logical end of the programs, the data base changes are committed. (The connection to SQL/DS is implicitly released on program termination.) Note that the INCLUDE SQLCA statement is also near the end of the programs. This is possible because the Assembler preprocessor is a two-pass operation.

Observe that all SQL statements must be preceded by "EXEC SQL". There is no trailing delimiter in Assembler.

The DS and DC statements for the host variables were determined by referring to Figure 46 at the end of this appendix; This figure gives the Assembler representation for each of the seven SQL/DS data types supported by Assembler programs. When you are coding your own applications you'll need to obtain the data types of the columns that your host variables interact with. This can be done either by consulting the person who created the table, or by querying the SQL/DS catalogs. The SQL/DS catalogs are described in the *SQL/Data System Planning and Administration for VM/SP* manual.

```

*****
*
*      SAMPLE PROGRAM FOR VM/SP   ARISASMC
*
*
*
*      PURPOSE:      THIS PROGRAM SERVES TWO PURPOSES:
*                    1. IT IS AN EXAMPLE FOR HOW TO IMBED SQL
*                      STATEMENTS IN AN ASSEMBLER PROGRAM.
*                    2. IT CAN BE USED TO TEST SOME BASIC SQL
*                      STATEMENTS FROM AN APPLICATION PROGRAM.
*
*
*      DESCRIPTION:  THIS PROGRAM GENERATES A SAMPLE ORDER FOR
*                    THE PARTS 315 AND 301 IF QONHAND IS LESS
*                    THAN 1000 AND 700 RESPECTIVELY AND IF
*                    QONORDER IS ZERO. THE TABLE QUOTATIONS
*                    IS UPDATED ACCORDINGLY. PART 315 IS
*                    ORDERED FROM THE COMPANY THAT SELLS IT
*                    FOR THE LOWEST PRICE, PART 301 FROM THE
*                    COMPANY WITH THE SHORTEST DELIVERY TIME.
*
*                    AT THE END OF THE PROGRAM PART 320 IS
*                    DELETED FROM THE DATA BASE.
*
*
*      PREREQUISITE: THE SQL/DS SAMPLE TABLES MUST BE CREATED
*                    AND LOADED.
*
*
*      OUTPUT PRODUCED: 1. AN EXECUTION BEGIN AND END MESSAGE IS
*                       PRINTED AND DISPLAYED AT THE BEGIN
*                       AND END OF PROGRAM EXECUTION.
*                       2. ALL TABLES ARE PRINTED WITH THEIR ORIGI-
*                         NAL CONTENTS.
*                       3. A SAMPLE ORDER IS PRINTED.
*                       5. THE CONTENTS OF ALL TABLES ARE PRINTED
*                         AFTER THE UPDATE / DELETE STEP.
*                       6. UNEXPECTED RETURN CODE:
*                         AN ERROR MESSAGE IS ISSUED TOGETHER
*                         WITH THE SQLCA-INFORMATION AND CHANGES
*                         ARE BACKED OUT.
*
*****
*
*      EJECT
*
*****
*+++++ SAMPLE PROGRAM ARISASMC ++++++
*****
*
*      PRINT NOGEN
*
SAMPA   START X'00'
        STM   R14,R12,D12(R13)   SAVE REGISTERS
        BALR  R7,0               LOAD BASE REGISTER
        USING *,R7,R11,R12      ESTABLISH ADDRESSABILITY
        B     S000              BRANCH AROUND CONSTANTS
*
*****
*      ++ DECLARE HOST VARIABLES
*****
*
*      EXEC SQL BEGIN DECLARE SECTION
*
PARTNO  DS      H                SMALLINTEGER
*
*      THE DESCRIPTION COLUMN IS DEFINED AS VARCHAR(24) IN THE INVENTORY
*      TABLE. SQL/DS WILL CONVERT THE DATA TO FIXED LENGTH AND TRUNCATE
*      ANYTHING OVER 10 CHARACTERS WHEN THE DESCR HOST VARIABLE IS USED.
*
DESCR   DS      CL10             CHARACTER
QONHAND DS      F                INTEGER
SUPPNO  DS      H                SMALLINTEGER

```

```

NAME      DS      CL15              CHARACTER
ADDRESS   DS      H,CL35           VARCHAR
TIME      DS      H                 SMALLINTEGER
QONORDER  DS      F                 INTEGER
PRICE     DS      PL6'999999999.99' DECIMAL(11,2)
ID        DC      CL8'SQLDBA'       USER ID
PASSW     DC      CL8'SQLDBAPW'     PASSWORD
*
      EXEC SQL END DECLARE SECTION
*
*
S000      LA      R11,XFFF(R7)       LOAD BASE REGISTER
          LA      R11,D1(R11)       LOAD BASE REGISTER
          LA      R12,XFFF(R11)     LOAD BASE REGISTER
          LA      R12,D1(R12)       LOAD BASE REGISTER
*
*****
*      ++ SET UP SAVE AREA POINTERS *
*****
          ST      R13,SAVE0+D4       STORE BACKWARD POINTER TO SAVEAREA
          LA      R9,SAVE0           R9:=ADDR(NEW SAVE AREA)
          ST      R9,D8(R13)        STORE FORWARD POINTER TO SAVE AREA
          LR      R13,R9             R13:=ADDR(NEW SAVE AREA)
*
*****
*      ++ OPEN PRINT FILE AND PRINT START MESSAGE *
*****
          OPEN    (PRFILE,(OUTPUT),CSFILE,(OUTPUT))
          OPEN    PRINTER AND CONSOLE FILE
          PUT     CSFILE,OUTAREA     TYPE 'SAMPLE PROGR. EXECUTING'
          MVI    OUTCNTR,SKIP3      SKIP3
          BAL    R9,PRINT           PRINT 'SAMPLE PROGR. EXECUTING'
*
*****
*      ++ GET VIRTUAL STORAGE FOR SQL/DS AND INITIALIZE IT TO ZERO *
*****
          LA      R1,7(0,0)         COMPUTE SIZE OF STORAGE IN DWORDS.
          A      R1,SQLDSIZ
          SRL    R1,3
          ST     R1,DSIZE
          LR     R0,R1
          DMSFREE DWORDS=(0)
          LTR    R15,R15            TEST IF RETURN CODE
          BZ     S005               NO RETURN CODE
          MVC    OUTAREA(L'MSG01),MSG01 MOVE MESSAGE TO OUTPUT AREA
          BAL    R9,PRINT           PRINT 'DMSFREE FAILED'
          B      SAMPTRM           TERMINATE
*
S005      LR      R6,R1             R6:=ADDR(DMSFREE AREA)
          USING  SQLDSECT,R6       ESTABLISH ADDRESSABILITY
          LR     R4,R0              R4:=LENGH OF DMSFREE SPACE
S010      MVI    D0(R1),X'00'      CLEAR THE AREA
          LA     R1,D1(R1)         INCREMENT R1
          BCT   R4,S010           LOOP
*
*
*      PROGRAM WILL IGNORE WARNINGS SINCE THEY WILL NOT AFFECT RESULTS *
      EXEC SQL WHENEVER SQLWARNING CONTINUE
      EXEC SQL WHENEVER SQLERROR GOTO SQLERR
      EXEC SQL WHENEVER NOT FOUND GOTO SQLERR
*
*

```

```

MVC MSG02C,FUNC1 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL CONNECT :ID IDENTIFIED BY :PASSW
*
*****
* ++ PRINT THE THREE TABLES *
*****
*
BAL R8,INVENT PRINT TABLE INVENTORY
BAL R8,QUOT PRINT TABLE QUOTATIONS
BAL R8,SUPPL PRINT TABLE SUPPLIERS
*
*****
* ++ FIND MINIMUM PRICE FOR PART NUMBER 315. *
* THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM PRICE OF *
* ALL OCCURRENCES OR PART NUMBER 315 WITH QONHAND LESS THAN 100*
* QONORDER ZERO. AS PRICE IS A COLUMN IN QUOTATIONS AND QONHAND*
* A COLUMN IN INVENTORY THE TWO TABLES HAVE TO BE LINKED *
* VIA A JOIN BETWEEN THE PART NUMBERS IN INVENTORY AND THOSE *
* IN QUOTATIONS. NO CURSOR IS USED BECAUSE THE STATEMENT RE- *
* TURNS ONLY ONE ROW. *
*****
MVC MSG02C,FUNC2 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL SELECT MIN(PRICE) INTO :PRICE
FROM INVENTORY, QUOTATIONS
WHERE INVENTORY.PARTNO = 315 AND
QONHAND < 1000 AND
INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
QONORDER = 0
*
*****
* ++ RETRIEVE DATA OF PART 315 FOR THE SAMPLE ORDER. *
* THE FOLLOWING SELECT STATEMENT RETRIEVES DATA THAT WILL BE *
* USED FOR PRINTING A SAMPLE ORDER FOR PART 315 WITH THE *
* LOWEST PRICE. THE STATEMENT CONTAINS TWO JOIN CONDITIONS *
* BECAUSE DATA FROM ALL THEE TABLES IS REQUIRED. *
*****
MVC MSG02C,FUNC3 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND, PRICE,
NAME, ADDRESS, QUOTATIONS.SUPPNO
INTO :PARTNO,:DESCR,:QONHAND,:PRICE,:NAME,:ADDRESS,
:SUPPNO
FROM INVENTORY, QUOTATIONS, SUPPLIERS
WHERE INVENTORY.PARTNO = 315 AND
INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
PRICE = :PRICE AND
QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO
*
*****
* ++ STORE OUTPUT OF ABOVE SQL STATEMENT INTO THE OUTPUT AREA *
* FOR PRINTING AND CONVERT THE DATA IF NECESSARY. *
*****
*
USING DSECT4,R10 ESTABLISH ADDRESSABILITY
LA R10,FLD1 R10:=ADDR(SECOND OUTPUT AREA)
MVC D4PART,=C'1000 ' NUMBER OF PARTS TO BE ORDERED
MVC D4DESCR(L'DESCR),DESCR MOVE DESCRIPTION INTO OUTP AREA
L R3,QONHAND R3:=QUANTITY ON HAND
CVD R3,CVDFLD CONVERT TO DECIMAL
UNPK D4QONH,CVDFLD UNPACK FOR PRINT
OI D4QONH+L'D4QONH-D1,XFO ERASE THE SIGN
MVC D4NAME,NAME SAVE NAME
LH R3,ADDRESS R3:=LENGTH OF ACTUAL ADDRESS
BCTR R3,R0 R3:=R3-1

```

```

S020    EX    R3,S020                ADJUST LENGTH FIELD IN MVC
        MVC    D4ADDR(L0),ADDRESS+L2  SAVE ADDRESS
        UNPK   D4PRICE,PRICE          UNPACK PRICE
        OI    D4PRICE+L'D4PRICE-D1,XF0 ERASE SIGN
        MVC    D4PRICE-D1(L'D4PRICE-L2),D4PRICE SPACE FOR DECIMAL POINT
        MVI    D4PRICE+L'D4PRICE-D3,C'.'  INSERT DECIMAL POINT
        MVC    PRICE1,PRICE           SAVE PRICE
        MP    PRICE1,=P'1000'         CALCULATE SUM
        UNPK   D4TOTAL,PRICE1         UNPACK TOTAL
        OI    D4TOTAL+L'D4TOTAL-D1,XF0 ERASE SIGN
        MVC    D4TOTAL-D1(L'D4TOTAL-L2),D4TOTAL SPACE FOR DECIMAL POINT
        MVI    D4TOTAL+L'D4TOTAL-D3,C'.'  INSERT DECIMAL POINT

```

```

*
*****
*      ++ UPDATE QONORDER FOR PART 315 IN TABLE QUOTATIONS.          *
*      1000 PARTS WILL BE ORDERED.                                    *
*****
*

```

```

        MVC    MSG02C,FUNC4          MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL UPDATE QUOTATIONS SET QONORDER = 1000
        WHERE PARTNO = :PARTNO AND
        QONORDER = 0 AND PRICE = :PRICE AND SUPPNO = :SUPPNO

```

```

*
*****
*      ++ FIND MINIMUM DELIVERY TIME FOR PART 301                    *
*      THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM DELIVERY   *
*      TIME OF ALL OCCURENCES OF PART NUMBER 301 WITH QONHAND LESS  *
*      THAN 700 AND QONORDER ZERO. AS DELIVERY TIME IS A COLUMN IN  *
*      QUOTATIONS AND QONHAND A COLUMN IN INVENTORY THE TWO TABLES *
*      HAVE TO BE LINKED VIA A JOIN BETWEEN THE PART NUMBERS IN    *
*      QUOTATIONS AND THOSE IN INVENTORY.                            *
*****
*

```

```

        MVC    MSG02C,FUNC5          MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL SELECT MIN(DELIVERY_TIME)
        INTO :TIME
        FROM INVENTORY, QUOTATIONS
        WHERE INVENTORY.PARTNO = 301 AND
        QONHAND < 700 AND
        INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
        QONORDER = 0

```

```

*
*****
*      ++ RETRIEVE DATA OF PART 301 FOR THE SAMPLE ORDER.          *
*      THE FOLLOWING SELECT STATEMENT RETRIEVES DATA THAT WILL BE  *
*      USED FOR PRINTING A SAMPLE ORDER FOR PART 301 WITH THE      *
*      LOWEST PRICE. THE STATEMENT CONTAINS TWO JOIN CONDITIONS     *
*      BECAUSE DATA FROM ALL THEE TABLES IS REQUIRED.             *
*****
*

```

```

        MVC    MSG02C,FUNC6          MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND, PRICE,
        NAME, ADDRESS, QUOTATIONS.SUPPNO
        INTO :PARTNO, :DESCR, :QONHAND, :PRICE, :NAME, :ADDRESS,
        :SUPPNO
        FROM INVENTORY, QUOTATIONS, SUPPLIERS
        WHERE INVENTORY.PARTNO = 301 AND
        INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
        DELIVERY_TIME = :TIME AND
        QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO

```

```

*
*****
*      ++ UPDATE QONORDER FOR PART 301 IN TABLE QUOTATIONS.          *
*      700 PARTS WILL BE ORDERED.                                    *
*****
*

```

```

MVC MSG02C,FUNC7 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL UPDATE QUOTATIONS SET QONORDER = 700
WHERE PARTNO = :PARTNO AND QONORDER = 0 AND
DELIVERY_TIME = :TIME AND SUPPNO = :SUPPNO
*
*
*****
* ++ WRITE A SAMPLE ORDER FOR PART 301 AND 315 *
*****
*
S035 MVI OUTCNTR,PAGE SET PRINTER CONTROL CHARACTER
MVC OUTAREA(L'MSG03),MSG03 MOVE MSG TO OUTPUTAREA
BAL R9,PRINT PRINT 'SAMPLE ORDER ...'
MVI OUTCNTR,SKIP3 SET PRINTER CONTROL CHARACTER
MVC OUTAREA(L'MSG04),MSG04 MOVE MSG TO OUTPUTAREA
BAL R9,PRINT PRINT 1. HEADING
MVC OUTAREA(L'MSG05),MSG05 MOVE MSG TO OUTPUTAREA
BAL R9,PRINT PRINT 2. HEADING
*
MVC OUTAREA,FLD1 MOVE OUTPUT OF FIRST SELECT STMT
MVI OUTCNTR,SKIP2 SET PRINTER CONTROL CHARACTER
BAL R9,PRINT PRINT OUTPUT OF FIRST SELECT STMT
*
*****
* ++ STORE OUTPUT OF ABOVE SQL STATEMENT INTO THE OUTPUT AREA *
* FOR PRINTING AND CONVERT THE DATA IF NECESSARY. *
*****
*
LA R10,OUTAREA R10:=ADDR(SECOND OUTPUT AREA)
MVC D4PART,=C' 700 ' NUMBER OF PARTS TO BE ORDERED
MVC D4DESCR(L'DESCR),DESCR MOVE DESCRIPTION INTO OUTP AREA
L R3,QONHAND R3:=QUANTITY ON HAND
CVD R3,CVDFLD CONVERT TO DECIMAL
UNPK D4QONH,CVDFLD UNPACK FOR PRINT
OI D4QONH+L'D4QONH-D1,XF0 ERASE THE SIGN
MVC D4NAME,NAME SAVE NAME
LH R3,ADDRESS R3:=ACTUAL LENGTH OF ADDRESS
BCTR R3,R0 R3:=R3-1
EX R3,S040 ADJUST LENGTH FIELD IN MVC
S040 MVC D4ADDR(L0),ADDRESS+L2 SAVE ADDRESS, OMIT LENGTH FIELD
UNPK D4PRICE,PRICE UNPACK PRICE
OI D4PRICE+L'D4PRICE-D1,XF0 ERASE SIGN
MVC D4PRICE-D1(L'D4PRICE-L2),D4PRICE SPACE FOR DECIMAL POINT
MVI D4PRICE+L'D4PRICE-D3,C'.' INSERT DECIMAL POINT
MP PRICE,=P'700' CALCULATE SUM
UNPK D4TOTAL,PRICE UNPACK PRICE
OI D4TOTAL+L'D4TOTAL-D1,XF0 ERASE SIGN
MVC D4TOTAL-D1(L'D4TOTAL-L2),D4TOTAL SPACE FOR DECIMAL POINT
MVI D4TOTAL+L'D4TOTAL-D3,C'.' INSERT DECIMAL POINT
MVI OUTCNTR,SKIP2 SET PRINTER CONTROL CHARACTER
BAL R9,PRINT PRINT OUTPUT OF SECOND SELECT STMT
*
*****
* ++ DELETE PART 320 FROM TABLE INVENTORY *
*****
*
MVC MSG02C,FUNC8 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL DELETE FROM INVENTORY WHERE PARTNO = 320
*
*****
* ++ DELETE PART 320 FROM TABLE QUOTATIONS *
*****
*
MVC MSG02C,FUNC9 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL DELETE FROM QUOTATIONS WHERE PARTNO = 320
*
*****
* ++ PRINT THE THREE TABLES *
*****

```



```

*****
*
MVC MSG07+L'MSG07-L'MSG16(L'MSG16),MSG16 'AFTER CHANGE'
MVC MSG08+L'MSG08-L'MSG16(L'MSG16),MSG16 'AFTER CHANGE'
BAL R8,INVENT PRINT TABLE INVENTORY
BAL R8,QUOT PRINT TABLE QUOTATIONS
BAL R8,SUPPL PRINT TABLE SUPPLIERS
*
*
* ++ COMMIT ALL CHANGES THAT HAVE BEEN MADE IN THE DATA BASE
*
MVC MSG02C,FUNC10 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL COMMIT WORK
*
*****
* ++ PRINT END MESSAGE, RESTORE REGISTERS AND TERMINATE *
*****
S050 MVC OUTAREA(L'MSG06),MSG06 MOVE MESSAGE TO OUTPUT AREA
PUT CSFILE,OUTAREA TYPE 'ARISASM COMPLETED SUCCESSF.'
MVI OUTCNTR,SKIP3 SET PRINTER CONTROL CHARACTER
BAL R9,PRINT PRINT 'ARISASM COMPLETED SUCCESSF.'
*
SAMPEND CLOSE (CSFILE,,PRFILE)
L R0,DSIZE
LR R1,R6
DMSFRET DWORDS=(0),LOC=(1)
SAMPTRM L R13,SAVE0+D4 R13:=ADDR(SAVE AREA)
LM R14,R12,D12(R13) RELOAD REGISTERS
BR R14 -- END OF PROGRAM --
*
*****
* +++++ THE FOLLOWING ROUTINE PRINTS THE TABLE INVENTORY +++++*
*****
EXEC SQL WHENEVER NOT FOUND CONTINUE
*
*****
* ++ PRINT HEADINGS FOR TABLE INVENTORY *
*****
INVENT MVI OUTCNTR,PAGE PRINTER CONTROL CHARACTER
MVC OUTAREA(L'MSG07),MSG07 MOVE MSG TO OUTPUT AREA
BAL R9,PRINT PRINT 'PRINTOUT OF TABLE INVENTORY'
*
USING DSECT1,R10 ESTABLISH ADDRESSABILITY
LA R10,OUTAREA R10:=ADDR(OUTAREA)
MVC D1PART,=C'PARTNO' MOVE TO OUTPUT AREA
MVC D1DESCR,=C'DESCRIPTION' MOVE TO OUTPUT AREA
MVC D1QONH,=C'QONHAND ' MOVE TO OUTPUT AREA
MVI OUTCNTR,SKIP3 SKIP 3
BAL R9,PRINT PRINT HEADING
*
MVC D1PART,UNDERSC MOVE TO OUTPUT AREA
MVC D1DESCR,UNDERSC MOVE TO OUTPUT AREA
MVC D1QONH,UNDERSC MOVE TO OUTPUT AREA
BAL R9,PRINT PRINT HEADING
*
*****
* ++ DECLARE AND OPEN CURSOR FOR SUBSEQUENT FETCH OF ALL ROWS *
* IN TABLE INVENTORY *
*****
EXEC SQL DECLARE C1 CURSOR FOR
SELECT PARTNO, DESCRIPTION, QONHAND
FROM INVENTORY
*

```

ORDER BY PARTNO

```

*
*      MVC   MSG02C, FUNC11      MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL OPEN C1
*
*****
*      ++ FETCH AND PRINT A ROW OF TABLE INVENTORY AND REPEAT UNTIL      *
*      RETURN CODE +100 (END OF DATA) COMES UP.                          *
*****
*
INVENT05 MVC   MSG02C, FUNC12      MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL FETCH C1 INTO :PARTNO, :DESCR, :QONHAND
*
*      CLC   SQLCODE, FD100      TEST IF END OF DATA
*      BE    INVENT20           END OF DATA
*
*      LH    R3, PARTNO          R3:=PARTNO
*      CVD   R3, CVDFLD          CONVERT TO DECIMAL
*      UNPK  D1PART, CVDFLD      UNPACK FOR PRINT
*      OI    D1PART+L'D1PART-D1, XFO ERASE THE SIGN
*      MVC   D1DESCR(L'DESCR), DESCR MOVE TO OUTPUT AREA
*      L     R3, QONHAND         R3:=QONHAND
*      CVD   R3, CVDFLD          CONVERT TO DECIMAL
*      UNPK  D1QONH, CVDFLD      UNPACK FOR PRINT
*      OI    D1QONH+L'D1QONH-D1, XFO ERASE THE SIGN
*
*      BAL   R9, PRINT           PRINT ROW
*      B     INVENT05           READ NEXT ROW
*
INVENT20 MVC   MSG02C, FUNC13      MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL CLOSE C1
*      BR    R8                 RETURN
*
*
*****
*      +++++ THE FOLLOWING ROUTINE PRINTS THE TABLE QUOTATIONS +++++
*****
*
*      ++ PRINT HEADINGS FOR TABLE QUOTATIONS
*****
*
QUOT      MVI   OUTCNTR, PAGE      PRINTER CONTROL CHARACTER
*      MVC   OUTAREA(L'MSG08), MSG08 MOVE MSG TO OUTPUT AREA
*      BAL   R9, PRINT           PRINT 'PRINTOUT OF TABLE QUOTATIONS'
*
*      USING DSECT3, R10          ESTABLISH ADDRESSABILITY
*      LA    R10, OUTAREA         R10:=ADDR(OUTAREA)
*      MVC   D3SUPPNO, =C'SUPPNO' MOVE TO OUTPUT AREA
*      MVC   D3PART, =C'PARTNO'   MOVE TO OUTPUT AREA
*      MVC   D3PRICE, =C'PRICE '  MOVE TO OUTPUT AREA
*      MVC   D3TIME, =C'DELIVERY TIME' MOVE TO OUTPUT AREA
*      MVC   D3ORDER, =C'QONORDER ' MOVE TO OUTPUT AREA
*      MVI   OUTCNTR, SKIP3       SKIP 3
*      BAL   R9, PRINT           PRINT HEADING
*
*      MVC   D3SUPPNO, UNDERSC    MOVE TO OUTPUT AREA
*      MVC   D3PART, UNDERSC      MOVE TO OUTPUT AREA
*      MVC   D3PRICE, UNDERSC     MOVE TO OUTPUT AREA
*      MVC   D3TIME, UNDERSC      MOVE TO OUTPUT AREA
*      MVC   D3ORDER, UNDERSC     MOVE TO OUTPUT AREA
*      BAL   R9, PRINT           PRINT HEADING
*
*****
*      ++ DECLARE AND OPEN CURSOR FOR SUBSEQUENT FETCH OF ALL ROWS      *
*      IN TABLE QUOTATIONS                                             *
*

```

```

*****
*
EXEC SQL DECLARE C2 CURSOR FOR
      SELECT SUPPNO, PARTNO, PRICE, DELIVERY_TIME, QONORDER
      FROM QUOTATIONS
      ORDER BY SUPPNO, PARTNO
*
      MVC   MSG02C,FUNC14      MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL OPEN C2
*
*****
*   ++ FETCH AND PRINT A ROW OF TABLE QUOTATIONS AND REPEAT UNTIL *
*   RETURN CODE +100 (END OF DATA) COMES UP. *
*****
*
QUOT05  MVC   MSG02C,FUNC15      MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL FETCH C2 INTO :SUPPNO, :PARTNO, :PRICE, :TIME, :QONORDER
*
      CLC   SQLCODE,FD100      TEST IF END OF DATA
      BE   QUOT20              END OF DATA
*
      LH   R3,SUPPNO           R3:=PARTNO
      CVD  R3,CVDFLD           CONVERT TO DECIMAL
      UNPK D3SUPPNO,CVDFLD     UNPACK FOR PRINT
      OI   D3SUPPNO+L'D3SUPPNO-D1,XF0 ERASE THE SIGN
      LH   R3,PARTNO           R3:=PARTNO
      CVD  R3,CVDFLD           CONVERT TO DECIMAL
      UNPK D3PART,CVDFLD     UNPACK FOR PRINT
      OI   D3PART+L'D3PART-D1,XF0 ERASE THE SIGN
      UNPK D3PRICE,PRICE      UNPACK PRICE
      OI   D3PRICE+L'D3PRICE-D1,XF0 ERASE THE SIGN
      MVC  D3PRICE-D1(L'D3PRICE-L2),D3PRICE SPACE FOR DECIMAL POINT
      MVI  D3PRICE+L'D3PRICE-D3,C'.' INSERT DECIMAL POINT
      LH   R3,TIME             R3:=TIME
      CVD  R3,CVDFLD           CONVERT TO DECIMAL
      UNPK D3TIME,CVDFLD     UNPACK FOR PRINT
      OI   D3TIME+L'D3TIME-D1,XF0 ERASE THE SIGN
      L    R3,QONORDER         R3:=QONORDER
      CVD  R3,CVDFLD           CONVERT TO DECIMAL
      UNPK D3ORDER,CVDFLD     UNPACK FOR PRINT
      OI   D3ORDER+L'D3ORDER-D1,XF0 ERASE THE SIGN
*
      BAL  R9,PRINT            PRINT ROW
      B    QUOT05              READ NEXT ROW
*
QUOT20  MVC   MSG02C,FUNC16      MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL CLOSE C2
      BR   R8                  RETURN
*
*
*****
*   +++++ THE FOLLOWING ROUTINE PRINTS THE TABLE SUPPLIERS +++++
*****
*
*****
*   ++ PRINT HEADINGS FOR TABLE SUPPLIERS *
*****
*
SUPPL   MVI   OUTCNTR,PAGE      PRINTER CONTROL CHARACTER
      MVC   OUTAREA(L'MSG09),MSG09 MOVE MSG TO OUTPUT AREA
      BAL  R9,PRINT            PRINT 'PRINTOUT OF TABLE SUPPLIERS'
*
      USING DSECT2,R10          ESTABLISH ADDRESSABILITY
      LA   R10,OUTAREA          R10:=ADDR(OUTAREA)
      MVC  D2SUPPNO,=C'SUPPNO' MOVE TO OUTPUT AREA

```

```

MVC D2NAME(L4),=C'NAME' MOVE TO OUTPUT AREA
MVC D2ADDR(L7),=C'ADDRESS' MOVE TO OUTPUT AREA
MVI OUTCNTR,SKIP3 PRINTER CONTROL CHARACTER
BAL R9,PRINT PRINT HEADING
*
MVC D2SUPPNO,UNDERSC MOVE TO OUTPUT AREA
MVC D2NAME,UNDERSC MOVE TO OUTPUT AREA
MVC D2ADDR,UNDERSC MOVE TO OUTPUT AREA
BAL R9,PRINT PRINT HEADING
*
*****
* ++ DECLARE AND OPEN CURSOR FOR SUBSEQUENT FETCH OF ALL ROWS *
* IN TABLE SUPPLIERS *
*****
*
EXEC SQL DECLARE C3 CURSOR FOR *
SELECT SUPPNO, NAME, ADDRESS *
FROM SUPPLIERS *
ORDER BY SUPPNO *
*
MVC MSG02C,FUNC17 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL OPEN C3
*
*****
* ++ FETCH AND PRINT A ROW OF TABLE SUPPLIERS AND REPEAT UNTIL *
* RETURN CODE +100 (END OF DATA) COMES UP. *
*****
*
SUPPL05 MVC MSG02C,FUNC18 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL FETCH C3 INTO :SUPPNO, :NAME, :ADDRESS
*
CLC SQLCODE,FD100 TEST IF END OF DATA
BE SUPPL20 END OF DATA
*
LH R3,SUPPNO R3:=PARTNO
CVD R3,CVDFLD CONVERT TO DECIMAL
UNPK D2SUPPNO,CVDFLD UNPACK FOR PRINT
OI D2SUPPNO+L'D2SUPPNO-D1,XFO ERASE THE SIGN
MVC D2NAME(L'NAME),NAME MOVE TO OUTPUT AREA
LH R3,ADDRESS R3:=LENGTH OF ACTUAL ADDRESS
BCTR R3,R0 R3:=R3-1
EX R3,SUPPL15 ADJUST LENGTH FIELD IN MVC
SUPPL15 MVC D2ADDR(L0),ADDRESS+L2 SAVE ADDRESS
*
BAL R9,PRINT PRINT ROW
B SUPPL05 READ NEXT ROW
*
SUPPL20 MVC MSG02C,FUNC19 MOVE FUNCTION IN CASE OF AN ERROR
EXEC SQL CLOSE C3
BR R8 RETURN
*
*****
*+++++++ THE FOLLOWING ROUTINE ELIMINATES LEADING ZEROES AND ++++++*
*+++++++ PRINTS ONE LINE ON THE PRINTER. ++++++*
*****
*
PRINT LA R1,L'OUTAREA R1:=LENGTH(OUTPUT AREA)
LA R3,OUTAREA R3:=ADDR(OUTPUT AREA)
*
PRZ05 CLI D0(R3),CB FIND FIRST NONBLANK CHARACTER
BNE PRZ15 FOUND
PRZ10 LA R3,D1(R3) R3:=ADDR(NEXT BYTE IN OUTAREA)
BCT R1,PRZ05 REPEAT UNTIL END OF OUTAREA
B PRINT1 PRINT LINE
*
PRZ15 CLI D0(R3),CO TEST IF LEADING ZERO
BNE PRZ20 NOT LEADING ZERO

```

```

        CLI   D1(R3),CB           TEST IF LAST ZERO
        BE    PRZ10              LAST ZERO, DON'T BLANK
        MVI   D0(R3),CB         ERASE LEADING ZERO
        B     PRZ10             TEST NEXT BYTE
*
PRZ20   LA    R3,D1(R3)         R3:=ADDR(NEXT BYTE IN OUTAREA)
        BCT  R1,PRZ25          REPEAT UNTIL END OF OUTAREA
        B     PRINT1           PRINT LINE
*
PRZ25   CLI   D0(R3),CB         TEST IF BLANK
        BNE  PRZ20             NOT BLANK
        B     PRZ10             GO, FIND NEXT NUMBER
*
PRINT1  PUT   PRFILE,OUTCNTR    PRINT LINE
        MVI  OUTCNTR,CB        PREPARE FOR NEXT INSTRUCTION
        MVC  OUTAREA,OUTAREA-D1 CLEAR OUTPUT AREA
        MVI  OUTCNTR,SKIP1     SET CONTROL CHARACTER FOR PRINTER
        BR   R9                RETURN
*
*****
*+++++++ THE FOLLOWING ROUTINE PRINTS THE SQLCA STRUCTURE ++++++*
*****
*
SQLERR  MVI   OUTCNTR,SKIP2     SKIP 2
        MVC  OUTAREA(L'MSG02),MSG02 MOVE MESSAGE TO OUTPUT AREA
        BAL  R9,PRINT          PRINT 'SQL ERROR OCCURED...'
*
*****
*      ++ PRINT SQLCODE *
*****
*
        L    R1,SQLCODE        R1:= ERROR CODE
        CVD  R1,CVDFLD         CONVERT SQLCODE TO DECIMAL
        UNPK MSG10B,CVDFLD     UNPACK SQLCODE FOR PRINT
        TM   MSG10B+L'MSG10B-D1,X10 TEST IF NEGATIVE
        BZ   PR05              POSITIVE ERROR CODE
        MVI  MSG10B,MINUS      MOVE MINUS SIGN BEFORE SQL CODE
PR05    OI   MSG10B+L'MSG10B-D1,XF0 ERASE ORIGINAL SIGN
        MVC  OUTAREA(L'MSG10),MSG10 MOVE MESSAGE TO OUTPUT AREA
        BAL  R9,PRINT          PRINT 'SQLCODE: XXXX'
*
*****
*      ++ PRINT SQLERRM *
*****
*
        MVC  MSG11B,SQLERRM+D2 MOVE SQLERRM, OMIT THE LENGH FIELD
        MVC  OUTAREA(L'MSG11),MSG11 MOVE MESSAGE TO OUTPUT AREA
        BAL  R9,PRINT          PRINT 'SQLERRM: XXXXXX'
*
*****
*      ++ PRINT SQLERRP *
*****
*
        MVC  MSG12B,SQLERRP    MOVE ROUTINE NAME INTO MESSAGE
        MVC  OUTAREA(L'MSG12),MSG12 MOVE MESSAGE TO OUTPUT AREA
        BAL  R9,PRINT          PRINT 'SQLERRP: XXXXXX'
*
*****
*      ++ PRINT SQLERRD *
*****
*
        LA   R0,SQLERRD+ERRDLEN R0:=END OF DIAGNOSTIC FIELDS
        LA   R1,SQLERRD        R1:=ADDR(FIRST DIAGNOSTIC FIELD)
        LA   R2,MSG13B        R2:=ADDR(DIAGNOSTIC AREA IN MESSAGE)
*
PR10   L     R3,D0(R1)         R3:= DIAGNOSTIC INFORMATION

```

```

CVD    R3,CVDFLD          CONVERT DIAGNOSTIC INFO TO DECIMAL
UNPK   D0(L4,R2),CVDFLD  UNPACK DIAGNOSTIC INFO FOR PRINT
TM     D3(R2),X10        TEST IF NEGATIVE
BZ     PR11              POSITIVE ERROR CODE
MVI    D0(R2),MINUS      MOVE MINUS SIGN BEFORE VALUE
PR11   OI    D3(R2),XF0   ERASE THE SIGN
MVI    D4(R2),CB         PUT BLANK BETWEEN VALUES
LA     R1,D4(R1)         R1:=ADDR(NEXT DIAGNOSTIC FIELD)
LA     R2,D5(R2)         R2:=ADDR(NEXT DIAG FLD IN MESSAGE)
CR     R1,R0             TEST IF ALL FLDS HAVE BEEN CONVERTED
BL     PR10             NO, TAKE NEXT FIELD
*
MVC    OUTAREA(L'MSG13),MSG13 MOVE MESSAGE TO OUTPUT AREA
BAL    R9,PRINT          PRINT 'SQLERRD: XXX XXX XXX ...'
*
*****
*      ++ PRINT SQLWARN                                     *
*****
*
CLI    SQLWARN0,CB       TEST FOR WARNINGS
BE     PR25              NO WARNINGS
MVC    MSG14B,SQLWARN    MOVE WARNINGS INTO MESSAGE
LA     R1,L11            R1:=LENGTH OF WARNING FIELDS
LA     R2,MSG14B         R2:=ADDR(MESSAGE)
*
PR15   CLI    D0(R2),CB   TEST IF WARNING EXISTS FOR THIS FLD
BNE    PR20              YES, WARNING
MVI    D0(R2),C'.'      MARK THE POSITION
PR20   BCT    R1,PR15     LOOP EIGHT TIMES
MVC    OUTAREA(L'MSG14),MSG14 MOVE MESSAGE TO OUTPUT AREA
BAL    R9,PRINT          PRINT 'SQLWARN: XXXXXXXXXXXX'
*
PR25   CLC    MSG02C,FUNC20 TEST IF ERROR DURING ROLLBACK
BE     PR30              TERMINATE TO AVOID LOOP
*
MVC    MSG02C,FUNC20    MOVE FUNCTION IN CASE OF AN ERROR
EXEC   SQL ROLLBACK WORK
*
PR30   MVC    OUTAREA(L'MSG15),MSG15 MOVE MESSAGE INTO OUTPUT AREA
PUT    CSFILE,OUTAREA   TYPE 'SAMPLE PROGRAM UNSUCCESSFUL'
MVI    OUTCNTR,SKIP3    SET PRINTER CONTROL CHARACTER
BAL    R9,PRINT          WRITE 'SAMPLE PROGRAM UNSUCCESSFUL'
B      SAMPEND          GO TO END OF JOB
*
*
*
*****
*+++++ EQUATES AND CONSTANTS +++++*
*****
*
R0     EQU    0          REGISTER
R1     EQU    1          REGISTER
R2     EQU    2          REGISTER
R3     EQU    3          REGISTER
R4     EQU    4          REGISTER
R5     EQU    5          REGISTER
R6     EQU    6          REGISTER
R7     EQU    7          REGISTER
R8     EQU    8          REGISTER
R9     EQU    9          REGISTER
R10    EQU    10         REGISTER
R11    EQU    11         REGISTER
R12    EQU    12         REGISTER
R13    EQU    13         REGISTER
R14    EQU    14         REGISTER
R15    EQU    15         REGISTER
*

```

D0	EQU	0	DISPLACEMENT
D1	EQU	1	DISPLACEMENT
D2	EQU	2	DISPLACEMENT
D3	EQU	3	DISPLACEMENT
D4	EQU	4	DISPLACEMENT
D5	EQU	5	DISPLACEMENT
D8	EQU	8	DISPLACEMENT
D12	EQU	12	DISPLACEMENT
*			
L0	EQU	0	LENGTH
L2	EQU	2	LENGTH
L4	EQU	4	LENGTH
L7	EQU	7	LENGTH
L11	EQU	11	LENGTH
*			
XFO	EQU	X'FO'	HEX NUMBER, MASK
X10	EQU	X'10'	HEX NUMBER, MASK
XFFF	EQU	X'FFF'	HEX NUMBER, MASK
*			
CB	EQU	C' '	CHAR, BLANK
C0	EQU	C'0'	CHAR, NULL
MINUS	EQU	C'-'	CHAR
SKIP1	EQU	C' '	PRINTER CONTR CHR 'SPACE 1 AND WRITE'
SKIP2	EQU	C'0'	PRINTER CONTR CHR 'SPACE 2 AND WRITE'
SKIP3	EQU	C'-'	PRINTER CONTR CHR 'SPACE 3 AND WRITE'
PAGE	EQU	C'1'	PRINTER CONTR CHR SKIP TO NEXT PAGE
*			
DSIZE	DS	F	
*			
F0	DC	F'0'	RETURN CODE
FD100	DC	A(100)	RETURN CODE
FD501	DC	A(501)	RETURN CODE
PRICE1	DS	PL6	SAVE AREA FOR PRICE
UNDERSC	DC	C'-----'	
*			
SAVE0	DS	32F	SAVE AREA
*			
FUNC1	DC	C'CONNECT ***	'
FUNC2	DC	C'SELECT MIN 315 ***	'
FUNC3	DC	C'SELECT 315 ***	'
FUNC4	DC	C'UPDATE 315 ***	'
FUNC5	DC	C'SELECT MIN 301 ***	'
FUNC6	DC	C'SELECT 301 ***	'
FUNC7	DC	C'UPDATE 301 ***	'
FUNC8	DC	C'DELETE INV 320 ***	'
FUNC9	DC	C'DELETE QUO 320 ***	'
FUNC10	DC	C'COMMIT WORK ***	'
FUNC11	DC	C'OPEN INV ***	'
FUNC12	DC	C'FETCH INV ***	'
FUNC13	DC	C'CLOSE INV ***	'
FUNC14	DC	C'OPEN QUO ***	'
FUNC15	DC	C'FETCH QUO ***	'
FUNC16	DC	C'CLOSE QUO ***	'
FUNC17	DC	C'OPEN SUPP ***	'
FUNC18	DC	C'FETCH SUPP ***	'
FUNC19	DC	C'CLOSE SUPP ***	'
FUNC20	DC	C'ROLLBACK WORK ***	'
*			
OUTCNTR	DC	C' '	PRINTER CONTROL CHARACTER
OUTAREA	DC	CL130'*** SAMPLE PROGRAM ARISASM EXECUTING ***'	
FLD1	DC	CL130' '	SECOND OUTPUTAREA
*			
	DS	0D	ALIGNMENT FOR NEXT FIELD
CVDFLD	DC	PL8'0'	FIELD USED FOR CVD INSTRUCTION
*			
MSG01	DC	C'*** DMSFREE FAILED ***'	

```

*
MSG02A DC C'*** SQL ERROR OCCURRED '
MSG02B DC C'FUNCTION = '
MSG02C DS CL(L'FUNC1)
ORG MSG02A
MSG02 DS CL(L'MSG02A+L'MSG02B+L'MSG02C)
*
MSG03 DC C'          * * *   S A M P L E   O R D E R   F O R   *
          P A R T S   3 0 1   A N D   3 1 5   * * * '
*
MSG04 DC C'NUMBER OF   DESCRIPTION   QUANTITY   COMPANY NAME   *
          COMPANY ADDRESS   PRICE PER   TOTAL '
MSG05 DC C' PARTS   ON HAND   UNIT   COSTS '
MSG06 DC C'*** SAMPLE PROGRAM ARISASMCM COMPLETED SUCCESSFULLY ****
          '
MSG07 DC C'          *** PRINTOUT OF TABLE INVENTORY UNCHANGED*
          *** '
MSG08 DC C'          *** PRINTOUT OF TABLE QUOTATION*
          S UNCHANGED *** '
MSG09 DC C'          *** PRINTOUT OF TABLE SUPPLIER*
          S UNCHANGED *** '
*
MSG10A DC C'SQLCODE: '
MSG10B DS CL4
ORG MSG10A
MSG10 DS CL(L'MSG10A+L'MSG10B)
SPACE
MSG11A DC C'SQLERRM: '
MSG11B DS CL70
ORG MSG11A
MSG11 DS CL(L'MSG11A+L'MSG11B)
*
MSG12A DC C'SQLERRP: '
MSG12B DS CL8
ORG MSG12A
MSG12 DS CL(L'MSG12A+L'MSG12B)
*
MSG13A DC C'SQLERRD: '
MSG13B DS CL5
MSG13C DS CL5
MSG13D DS CL5
MSG13E DS CL5
MSG13F DS CL5
MSG13G DS CL5
ORG MSG13A
MSG13 DS CL(L'MSG13A+L'MSG13B+L'MSG13C+L'MSG13D+L'MSG13E+L'MSG13F*
          +L'MSG13G)
*
MSG14A DC C'SQLWARN: '
MSG14B DS CL11
ORG MSG14A
MSG14 DS CL(L'MSG14A+L'MSG14B)
*
MSG15 DC C'*** EXECUTION OF SAMPLE PROGRAM ARISASMCM UNSUCCESSFUL *
          *** '
*
MSG16 DC C'AFTER MODIFICATION *** '
*
CSFILE DCB DDNAME=SYSCON,DSORG=PS,LRECL=130,MACRF=PM,RECFM=F
SPACE
PRFILE DCB DDNAME=SYSPRT,DSORG=PS,LRECL=131,MACRF=PM,RECFM=FBA
SPACE
EXEC SQL INCLUDE SQLCA
ERRDLEN EQU 6*L'SQLERRD
SPACE 3
DSECT1 DSECT

```


D1PART	DS	CL20	PARTNUMBER FIELD WITHIN INVENTORY
	DS	CL6	
	DS	CL2	
D1DESCR	DS	CL11	DESCRIPTION FIELD WITHIN INVENTORY
	DS	CL2	
D1QONH	DS	CL11	QONHAND FIELD WITHIN INVENTORY
	SPACE	3	
DSECT2	DSECT		
	DS	CL20	
D2SUPPNO	DS	CL6	SUPPNO FIELD WITHIN SUPPLIERS
	DS	CL2	
D2NAME	DS	CL15	NAME FIELD WITHIN SUPPLIERS
	DS	CL2	
D2ADDR	DS	CL35	ADDRESS FIELD WITHIN SUPPLIERS
	SPACE	3	
DSECT3	DSECT		
	DS	CL20	
D3SUPPNO	DS	CL6	SUPPNO FIELD WITHIN QUOTATIONS
	DS	CL2	
D3PART	DS	CL6	PARTNUMBER FIELD WITHIN QUOTATIONS
	DS	CL2	
D3PRICE	DS	CL7	PRICE FIELD WITHIN QUOTATIONS
	DS	CL2	
D3TIME	DS	CL13	TIME FIELD WITHIN QUOTATIONS
	DS	CL2	
D3ORDER	DS	CL11	QONORDER FIELD WITHIN QUOTATIONS
*			
DSECT4	DSECT		
	DS	CL3	
D4PART	DS	CL4	PARTS FIELD IN OUTPUT AREA
	DS	CL5	
D4DESCR	DS	CL11	DESCRIPTION FIELD IN OUTPUT AREA
	DS	CL3	
D4QONH	DS	CL6	QONHAND FIELD IN OUTPUT AREA
	DS	CL5	
D4NAME	DS	CL15	NAME FIELD IN OUTPUT AREA
	DS	CL2	
D4ADDR	DS	CL35	ADDRESS FIELD IN OUTPUT AREA
	DS	CL2	
D4PRICE	DS	CL8	PRICE FIELD IN OUTPUT AREA
	DS	CL2	
D4TOTAL	DS	CL8	TOTAL COST FIELD IN OUTPUT AREA
	END		

Rules for Using SQL in Assembler

This section lists, for your reference, all the rules for embedding SQL statements within an Assembler program.

Note: OPSYN and ICTL Assembler statements may not be used.

Declaring Host Variables

The following example shows an SQL declare section for an Assembler program:

```
Col. 1                                     Col. 72
|                                           |
|                                           |
|                                           |
LABEL EXEC SQL BEGIN DECLARE SECTION
AA      DS      F
BB      DC      H'3'          comment
* comment card or
* comment section
CC      DC      CL80'xxxx.....xxxx*'
XYZ     DSECT
DD      DS      D
EE      DS      CL5
FF      DS      H,CL40
        ORG     FF
GG      DS      H
HH      DS      CL40          comment
*                                           continued comment
II      DS      PL5
JJ      DC      PL5'123.45'
KK      DS      0H
LL      DS      CL12
LABEL2  EXEC          SQL  END DECLARE SECTION  comment
```

The above example illustrates the following rules:

1. All assembler variables that are to be used in SQL statements must be declared, and their declarations must appear within one or more sections that begin with

```
EXEC SQL BEGIN DECLARE SECTION
```

and end with

```
EXEC SQL END DECLARE SECTION
```

Each of these two statements must be totally contained on one line. Note that there is no semicolon delimiter at the end of the SQL statements. There may be a label on either of the statements, and comments are allowed after the statements.

2. Comments are allowed on any statement within the host variable declare section. Also, comment line images (* in column 1) are allowed in the declare section.
3. The Assembler preprocessor processes the line(s) in the declare section as follows:
 - a. If there is no label, the preprocessor ignores the line and goes on to the next.
 - b. If there is a label, but the opcode is not DS or DC, the preprocessor ignores the line and goes on to the next.
 - c. If there is a label and a DS or DC opcode, the operand is checked. The operand must be an acceptable data type as shown in the Figure 46 on page 446. Here are some examples:

```

F
F'5'
H
H'100'
CL255
CL5'ABCDE'
H,CL5
H'5',CL5'ABCDE'
D
D'2.5E10'
PL2
PL5'123.45'
H,CL32767

```

The first character of the operand may also be 0 and used as follows:

```

OH
OF
OD
OC

```

In this case, the line is ignored and the next line is processed.

If there are no errors at this stage, the variable is validly defined as a host variable. If there are errors, the line is flagged as an error, and the next line is processed.

4. Continuations are allowed by using a non-blank character in column 72 and beginning the next line in column 16.
5. The declare section can be anywhere that a normal DS or DC can be used. Because the SQL/DS assembler preprocessor is a two-pass operation, the declare section can come after the SQL statements that use the host variables.
6. There can be more than one host variable declare section in a program.

Embedding SQL Statements

The following are the rules for embedding SQL statements within Assembler programs:

1. Each SQL statement must be preceded by “EXEC SQL”, which must be on the same line. Only blanks can appear between the “EXEC” and “SQL”. There must not be a semicolon (;) delimiter on the SQL statement.
2. The first line of an SQL statement can have a label beginning in column 1. If there is no label, the SQL statement must begin in column 2 or greater.
3. Continuations are allowed by placing a non-blank character in column 72 and beginning the next line in column 16.
4. No comments or comment lines are allowed within an SQL statement. Any comments are considered part of the SQL statement.
5. Avoid using labels or variable names that begin with SQL, ARI, or RDI. Also avoid names beginning with PID, PBC, PA, PB, PC, PD, PE, PL, or PN where these letters are followed by numbers. These names may conflict with names generated by the assembler preprocessor.

Using the INCLUDE Command

To include external secondary input, specify:

```
EXEC SQL INCLUDE text-name
```

at the point in the source code where the secondary input is to be included. Text-name is the filename of a CMS file with an “ASMCOPY” filetype, located on a CMS minidisk accessed by the user.

The INCLUDE command must be completely contained on one line. There may be a label on the command, and comments are allowed after the command.

Using Host Variables in SQL Statements

When you place host variables within an SQL statement, you must precede each such variable with a colon (:). The colon distinguishes the host variables from the SQL identifiers (such as PARTNO). When the same variable is used outside of an SQL statement, do not use a colon.

A host variable can represent a data value, but not an SQL identifier. For example, you cannot assign a character constant, such as ‘MUSICIANS’, to a host variable, and then use that host variable in a CREATE TABLE statement to represent the table name. This pseudo-code sequence is invalid:

```
TT = 'MUSICIANS'  
CREATE TABLE :TT (NAME ...
```

Incorrect

Using Double-Byte Character Set (DBCS) Constants

The Assembler Preprocessor does not support either DBCS constants or host program variables except through dynamically defined statements. With dynamically defined statements you can access DBCS data, using an SQLDA for question mark (?) parameters and using DBCS-type codes defined for the SQLDA structure. In addition, hexadecimal constants can be used for DBCS data representation in SQL statements.

If the DBCS option is set to YES, embedded SQL statements can contain character string constants and identifiers with DBCS characters enclosed by so and si.

SQL Error Handling

There are two ways to declare the return code structure (called SQLCA):

1. You may write:

```
EXEC SQL INCLUDE SQLCA
```

in your source program. The SQL/DS preprocessor replaces this with a declaration of the SQLCA structure.

2. You may declare the SQLCA directly as shown in Figure 44.

SQLCA	DS	0F
SQLCAID	DS	CL8
SQLCABC	DS	F
SQLCODE	DS	F
SQLERRM	DS	H, CL70
SQLERRP	DS	CL8
SQLERRD	DS	6F
SQLWARN	DS	0C
SQLWARN0	DS	C
SQLWARN1	DS	C
SQLWARN2	DS	C
SQLWARN3	DS	C
SQLWARN4	DS	C
SQLWARN5	DS	C
SQLWARN6	DS	C
SQLWARN7	DS	C
SQLWARN8	DS	C
SQLWARN9	DS	C
SQLWARNA	DS	C
SQLTEXT	DS	CL5

Figure 44. SQLCA Structure (in Assembler)

You must not declare the SQLCA within the SQL declare section. The meaning of the fields is explained under "Error Handling" on page 202.

Dynamic SQL Statements in Assembler

An SQLDA structure may be required for dynamically executed SQL statements. There are two ways to declare the SQLDA structure:

1. You may write:

```
EXEC SQL INCLUDE SQLDA
```

in your source program. The SQL/DS preprocessor replaces this with a declaration of the SQLDA structure.

2. You may declare the SQLDA directly as shown in Figure 45.

SQLDA	DSECT	
SQLDAID	DS	CL8
SQLDABC	DS	F
SQLN	DS	H
SQLD	DS	H
SQLVAR	DS	0F
SQLVARN	DSECT	
SQLTYPE	DS	H
SQLLEN	DS	0H
SQLPRSCN	DS	CL1
SQLSCALE	DS	CL1
SQLDATA	DS	A
SQLIND	DS	A
SQLNAME	DS	H,CL30
&SYSECT	CSECT	

Figure 45. SQLDA Structure (in Assembler)

The SQLDA structure must not be declared within an SQL declare section. When you specify INCLUDE SQLDA, the Assembler preprocessor generates a CSECT statement at the end of the SQLDA. This CSECT is generated with the name of the CSECT currently active in your program.

You must not specify a constant string on a PREPARE or EXECUTE IMMEDIATE statement. You can only specify a variable defined as a variable-length character string:

```
EXEC SQL PREPARE S1 FROM :STRING1
EXEC SQL EXECUTE IMMEDIATE :STRING1
      .
      .
EXEC SQL BEGIN DECLARE SECTION
STRING1 DS H,CLxxxxx (xxxxx <= 8192)
EXEC SQL END DECLARE SECTION
```

The halfword of STRING1 must contain the length of the string, and the character portion must contain the string itself when the PREPARE or EXECUTE IMMEDIATE is executed.

Data Types

Description	SQL/DS Keyword	Equivalent Assembler Declaration
A binary integer of 31 bits, plus sign.	INTEGER	F
A binary integer of 15 bits, plus sign.	SMALLINT	H
A packed decimal number, precision m, scale n ($1 \leq m \leq 15$ and $0 \leq n \leq m$). In storage the number occupies an even number of bytes up to a maximum of 8 bytes. Precision is the total number of digits. Scale is the number of those digits that are to the right of the decimal point.	DECIMAL(m[,n])	PLp ['decimal constant'] The precision is $2p-1$ and the scale is that of the decimal constant. If the constant is not specified, the scale is 0.
A double-precision (8- byte) floating point number, in standard System/370 floating point format.	FLOAT	D
A fixed-length character string of length n where $n \leq 254$.	CHAR(n)	CLn
A varying-length character string of maximum length n, where $n \leq 254$. (Only the actual length is stored in the data base.)	VARCHAR(n)	H,CLn
A varying-length character string of maximum length 32767 bytes, subject to certain usage limitations.	LONG VARCHAR	H,CLn

Figure 46. SQL/DS Data Types for Assembler

Note: Double-Byte Character Set (DBCS) host variables are not supported by the SQL/DS Assembler preprocessor.

Reentrant Programs

A reentrant program has the characteristic of dynamic allocation of space for data and save areas. This reentrant characteristic can be employed in Assembler programs. In this case, the data and save areas are allocated in a calling (driver) program and passed to a called (reentrant) program as parameters. Storage for these areas need not be allocated in the called program.

A convenient use for reentrancy is the use of an SQLDA-like structure declared as a DSECT in the calling program. This, in combination with an INCLUDE SQLDA statement in the called program, permits the passing back of values, extracted by a SELECT/FETCH in the called program, in a clean and simple manner. A DESCRIBE statement can be used by the called program to fill the SQLDA-like structure or it can be hand-filled in the driver program. Other SQL statements (i.e. INSERT, DELETE, UPDATE) utilize single data locations to communicate SQLCODES.

If statement results, other than the SQLCODE, are desired, an SQLCA-like structure can be allocated in the driver program. However, unlike the SQLDA-like structure allocation by a DSECT, the fields of the SQLCA-like structure must be hard-coded into the driver. An INCLUDE SQLCA statement is then required in the called program. SQLCA communication between the 2 programs can be achieved by passing the address of the first field of the SQLCA-like structure to the reentrant program. Then by knowing the length of the SQLCA structure, the other fields can be addressed.

The following are skeleton programs illustrating the use of the SQLDA-like structure and a single data location for communicating SQLCODEs. The reentrant example illustrates only a FETCH statement. If more than one "action" statement (INSERT, DELETE, etc.) is used, then various flags are needed to direct access to the individual operations. The required modifications to include an SQLCA-like structure follow these skeletons.


```

Driver  CSECT
* Standard Linkage Conventions
      .
      .
Qstring DC   H,'57',CL57'SELECT DESCRIPTION FROM INVENTORY WHERE      X
          QONHAND < 100'
*
* Single data location to be passed to reentrant program
  LA     R4,1
  ST     R4,Code   Flag to monitor when no more entries satisfy
                  search condition; also used to indicate
                  whether first call to reentrant program.
* Block of storage (SQLDA-like) to be passed to reentrant program
  LA     R4,Space
  USING  Locda
* Fill structure (or can be filled by DESCRIBE)
      .
      .
      LA   R7,Outarea+1   Address where result will be stored
      ST   R7,Locdata
      LA   R7,Indaddr     Address where indicator value will be stored
      ST   R7,Locind
      .
      .
* Call reentrant program -- loop needed to handle cursor operations
Loop   DS   0H
      CLC  Code,F100
      BE   Final
* Blank out output area for next result
      .
      .
      LA   R1,Parmlist
      L    R15,=V(Reentrant)
      BALR R14,R15
*
* Test indicator values -- move value into Outarea if null
      .
      .
* Print results -- use data conversion if output not in char form
      .
      .
      PUT  Printer
      CLC  Code,F0
      BM   Errchk
      B    Loop
Errchk DS   0H
* Print error messages
      .
      .
Final  DS   0H
* Restore registers
      .
      .
* Declare section
      .
      .
Save0  DS   32F   Areas to save registers upon calls

```

```

Save1    DS    32F
Code     DS    F
Parmlist DC    A(Qstring)
         DC    A(Space)
         DC    A(Code)
         DC    A(Outarea)
         DC    A(Save1)
Outarea  DS    CL80
Indaddr  DS    F
Space    DS    CL500
Locda    DSECT
Loclaid  DS    CL8
         .
         .
Locdata  DS    A
Locind   DS    A
Locname  DS    H,CL30
         .
         .
         END  Driver

```

Reentrant CSECT

* Standard Linkage Conventions. Use save area passed as parameter.

```
.
.
* Get addresses of parameters
  LR R10,R1
  L  R4,8(R10)   Code address
  L  R6,12(R10)  Outarea address
* Get storage for host variables
.
.
* Check if first call to this program
  CLC 8(4,R10),F0
  BE  Next
  EXEC SQL CONNECT ...
.
.
  MVC QSTRING(82),0(R10)  Parm string moved into host var
  EXEC SQL PREPARE S1 FROM :QSTRING
  CLC  SQLCODE,F0
  BNE Goback
  EXEC SQL DECLARE C1 CURSOR FOR S1
.
.
  EXEC SQL OPEN C1
.
.
Next   DS   0H
  EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
  CLC  SQLCODE,F100
  BE  Done
  CLC  SQLCODE,F0
  BE  Goback
.
.
Done   DS   0H
  EXEC SQL CLOSE C1
  CLC  SQLCODE,F0
  BE  Final
* Move appropriate SQLCODE into parameter position in Final or Goback
Final  DS   0H
.
.
Goback B    Exit
      DS   0H
.
.
Exit   DS   0H
* Restore registers and return
.
.
* Declare section
  EXEC SQL INCLUDE SQLDA
.
.
  END Reentrant
```

To include SQLCA communication the following is needed:

* Driver program DECLARE section

```
.  
. .  
Parmlist DC A(Qstring)  
          DC A(Space)  
          DC A(Code)  
          DC A(Outarea)  
          DC A(Lid)  
Locca    DS 0F  
Lid      DS CL8  
Lbc      DS F  
Lcode    DS F  
Lerrm    DS H,CL70  
Lerrp    DS CL8  
Lerrd    DS 6F  
Locwarn  DS 0C  
Warn0    DS CL1  
.  
.  
Warna    DS CL1  
Text     DS CL8
```

* In the Reentrant program

```
.  
. .  
L      R9,16(R1)  Address of 1st field in SQLCA-like structure  
.  
.  
MVC   12(4,R9),SQLCODE  To return SQLCODE  
                          To return any field of SQLCA modify  
                          offset. (Offset in given stmt is 12)
```

* Declare section
EXEC SQL INCLUDE SQLCA



Appendix F. FORTRAN Considerations

This appendix contains a sample program that illustrates the use of SQL within FORTRAN. Following the sample are specific rules for using SQL in FORTRAN.

ARISFTN -- FORTRAN Sample Program

ARISFTN is a FORTRAN sample program that is provided with SQL/DS. The source code of this program begins on the next page. You can learn most of the rules for using SQL within FORTRAN just by scanning through the program. Note, in particular, how the program satisfies the requirements of the application prolog and epilog. Near the beginning of the program all the host variables are declared, error handling is defined, and a connection is established with SQL/DS. Near the logical end of the program, the data base changes are committed. (The connection to SQL/DS is implicitly released on program termination.)

Notice that all SQL statements must be preceded by EXEC SQL. There is no trailing delimiter in FORTRAN.

The data description statements for the host variables were determined by referring to Figure 49 on page 466. That figure gives the FORTRAN representation for each of the seven SQL/DS data types that are supported by FORTRAN program (the DBCS data types are not supported in FORTRAN). When you are coding your own applications you'll need to obtain the data types of the columns that your host variables interact with. This can be done either by consulting the person who created the table, or by querying the SQL/DS catalogs. The SQL/DS catalogs are described in the *SQL/Data System Planning and Administration for VM/SP* manual.

```

*****
* SAMPLE PROGRAM      ARISFTN
*
* PURPOSE:           THIS PROGRAM SERVES TWO PURPOSES:
*                   1. IT IS AN EXAMPLE ON HOW TO IMBED SQL
*                   STATEMENTS IN A FORTRAN PROGRAM.
*                   2. IT CAN BE USED TO TEST SOME BASIC SQL
*                   FUNCTIONS FROM AN APPLICATION PROGRAM.
*
* DESCRIPTION:       THIS PROGRAM GENERATES A SAMPLE ORDER FOR
*                   THE PARTS 315 AND 316 IF QONHAND IS LESS
*                   THAN 1000 AND 700 RESPECTIVELY, AND IF
*                   QONORDER IS ZERO. THE TABLE QUOTATIONS
*                   IS UPDATED ACCORDINGLY. PART 315 IS ORDERED
*                   FROM THE COMPANY THAT SELLS IT FOR THE
*                   THE LOWEST PRICE, PART 316 FROM THE COMPANY
*                   WITH THE SHORTEST DELIVERY TIME.
*
*                   AT THE END OF THE PROGRAM PART 323 IS DELETED
*                   FROM THE DATA BASE.
*
* PREREQUISITE:     SUCCESSFUL EXECUTION OF THE COMMANDS IN
*                   ARISAMDB VIA THE DBS UTILITY.
*
* OUTPUT PRODUCED:  1. AN EXECUTION BEGIN AND END MESSAGE IS
*                   PRINTED AT BEGIN AND END OF PROGRAM
*                   EXECUTION.
*                   2. ALL TABLES ARE PRINTED WITH THEIR ORIGI-
*                   NAL CONTENTS.
*                   3. A SAMPLE ORDER IS PRINTED.
*                   4. THE CONTENTS OF THE TABLES ARE PRINTED
*                   AFTER ALL UPDATES AND DELETES ARE MADE.
*                   5. UNEXPECTED RETURN CODES:
*                   AN ERROR MESSAGE IS ISSUED TOGETHER WITH
*                   THE SQLCA-INFORMATION AND CHANGES BACKED
*                   OUT.
*
*****
PROGRAM TEST1
INTERNAL PROGRAM VARIABLES

CHARACTER*40      STMT
REAL*8           TPRCE1
REAL*8           TPRCE2

*
EXEC SQL INCLUDE SQLCA
*****
* ESTABLISH HOST VARIABLES
*****
EXEC SQL BEGIN DECLARE SECTION
INTEGER          PARTNO*2, QONHND, SUPPNO*2, TIME*2
* THE DESCRIPTION COLUMN IN THE INVENTORY TABLE IS VARCHAR(24)
* SQL/DS WILL CONVERT IT TO FIXED LENGTH AND TRUNCATE LONGER
* DATA WHENEVER THE DESCR HOST VARIABLE IS USED.
CHARACTER        DESCR*10, NAME*15, ADR*35
REAL*8           PRICE
CHARACTER*8      ID, PASS
EXEC SQL END DECLARE SECTION

DATA ID          /'SQLDBA '/
DATA PASS        /'SQLDBAPW' /
*****
* ERROR HANDLING
*****

```

```

*
* THIS PROGRAM WILL IGNORE WARNINGS AS THEY WILL NOT AFFECT RESULTS
  EXEC SQL WHENEVER SQLWARNING CONTINUE
  EXEC SQL WHENEVER SQLERROR GOTO 100
  EXEC SQL WHENEVER NOT FOUND GOTO 100
*****
* DISPLAY START MESSAGE *
*****
  PRINT *, ('SAMPLE PROGRAM ARISFTN STARTED')
*****
* START PROGRAM *
*****
  STMT = 'EXEC SQL CONNECT '
  SQLCOD = 0000
  EXEC SQL CONNECT :ID IDENTIFIED BY :PASS
  CONTINUE
*****
* PRINT TABLES *
*****
5  FORMAT ('1',TR10,A45)
  WRITE (6,5) '*** PRINTOUT OF TABLE INVENTORY UNCHANGED ***'
  CALL PRINT1
15 FORMAT ('1',TR10,A46)
  WRITE (6,15) '*** PRINTOUT OF TABLE QUOTATIONS UNCHANGED ***'
  CALL PRINT2
25 FORMAT ('1',TR10,A45)
  WRITE (6,25) '*** PRINTOUT OF TABLE SUPPLIERS UNCHANGED ***'
  CALL PRINT3
*****
* FIND MINIMUM PRICE FOR PART #315 *
*
* THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM PRICE OF ALL
* OCCURENCES OF PART #315 WITH QONHAND LESS THAN 1000, AND
* QONORDER 0. AS PRICE IS A COLUMN IN QUOTATIONS, AND QONHAND A
* COLUMN IN INVENTORY, THE TWO TABLES HAVE TO BE LINKED VIA A JOIN
* BETWEEN THE PART NUMBERS IN INVENTORY AND THOSE IN QUOTATIONS.
* A CURSOR IS DECLARED FOLLOWED BY A 'OPEN' CURSOR, 'FETCH' CURSOR
* AND 'CLOSE' CURSOR TO RETURN THE ROW.
*
*****
  STMT = 'DECLARE CURSOR S1 FOR MIN(PRICE) 315'
  EXEC SQL DECLARE S1 CURSOR FOR SELECT MIN(PRICE)
1    FROM INVENTORY, QUOTATIONS
2    WHERE INVENTORY.PARTNO = 315 AND
3          QONHAND < 1000 AND
4          INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
5          QONORDER = 0
  STMT = 'OPEN S1 FOR MIN(PRICE) 315 '
  EXEC SQL OPEN S1
  STMT = 'FETCH S1 FOR MIN(PRICE) 315'
  EXEC SQL FETCH S1
1    INTO :PRICE
  STMT = 'CLOSE S1 FOR MIN(PRICE) 315'
  EXEC SQL CLOSE S1
*****
* RETRIEVE DATA FOR ORDER AND *
* UPDATE QONORDER FOR PART #315, TABLE QUOTATIONS *
*****
  TPRCE1 = 1000 * PRICE
  STMT = 'DECLARE CURSOR S2 FOR PART #315'

  EXEC SQL DECLARE S2 CURSOR FOR
1    SELECT INVENTORY.PARTNO, DESCRIPTION, QONHAND,
2    PRICE, NAME, ADDRESS, QUOTATIONS.SUPPNO
3    FROM INVENTORY, QUOTATIONS, SUPPLIERS

```



```

4      WHERE INVENTORY.PARTNO = 315 AND
5          INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
6          PRICE = :PRICE AND
7          QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO

      STMT = 'OPEN FOR PART #315          '
      EXEC SQL OPEN S2
      STMT = 'FETCH FOR PART #315      '
      EXEC SQL FETCH S2
1      INTO :PARTNO, :DESCR, :QONHND, :PRICE, :NAME, :ADR,
2          :SUPPNO
      STMT = 'CLOSE FOR PART #315      '
      EXEC SQL CLOSE S2
      STMT = 'UPDATE QUOTATIONS        '
      EXEC SQL UPDATE QUOTATIONS SET QONORDER = 1000
1      WHERE PARTNO = :PARTNO AND
2          QONORDER = 0 AND
3          PRICE = :PRICE AND
4          SUPPNO = :SUPPNO

      CONTINUE
*****
*          PRINT SAMPLE ORDER WITH RESULTS OF PART #315          *
*****
45     FORMAT ('1', TR10, A27)
      WRITE (6, 45) 'SAMPLE ORDER          PAGE'
      PRINT *, (' NUMBER OF      DESCRIPTION      QUANTITY      COMPANY NAME      C
1COMPANY ADDRESS
      PRINT *, (' PARTS
1          UNITS      COSTS')
55     FORMAT (' ', TR2, A4, TR6, A10, I11, TR6, A15, TR1, A35, TR3, F7.2,
1TR4, F7.2)
      WRITE (6, 55) '1000', DESCR, QONHND, NAME, ADR, PRICE, TPRCE1
*****
*          FIND MINIMUM DELIVERY TIME FOR PART #316          *
*
* THE FOLLOWING SELECT STATEMENT RETURNS THE MINIMUM DELEVERY *
* TIME OF ALL OCCURRENCES OF PART #316 WITH QONHAND LESS THAN 700 *
* AND QONORDER 0. AS DELIVERY TIME IS A COLUMN IN QUOTATIONS *
* AND QONHAND A COLUMN IN INVENTORY, THE TWO TABLES HAVE TO BE *
* LINKED VIA A JOIN BETWEEN THE PART NUMBERS IN QUOTATIONS AND *
* THOSE IN INVENTORY. *
*
*****
      STMT = 'DECLARE CURSOR S3 FOR MIN(DELIVERY TIME) '

      EXEC SQL DECLARE S3 CURSOR FOR SELECT MIN(DELIVERY_TIME)
1      FROM INVENTORY, QUOTATIONS
2      WHERE INVENTORY.PARTNO = 316 AND
3          QONHAND < 700 AND
4          INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
5          QONORDER = 0
      STMT = 'OPEN MIN(DELIVERY TIME)          '
      EXEC SQL OPEN S3
      STMT = 'FETCH MIN(DELIVERY TIME)        '
      EXEC SQL FETCH S3
1      INTO :TIME
      STMT = 'CLOSE MIN(DELIVERY TIME)        '
      EXEC SQL CLOSE S3
      CONTINUE
*****
*          RETRIEVE DATA OF PART #316 FOR THE ORDER          *
*          AND *
*          UPDATE QONORDER FOR PART #316          *
*****
      STMT = 'DECLARE CURSOR S4 FOR PART #316          '
      EXEC SQL DECLARE S4 CURSOR FOR SELECT INVENTORY.PARTNO,

```

```

1      DESCRIPTION, QONHAND,
2      PRICE, NAME, ADDRESS
3      FROM INVENTORY, QUOTATIONS, SUPPLIERS
4      WHERE INVENTORY.PARTNO = 316 AND
5            INVENTORY.PARTNO = QUOTATIONS.PARTNO AND
6            DELIVERY_TIME = :TIME AND
7            QUOTATIONS.SUPPNO = SUPPLIERS.SUPPNO
      STMT = 'OPEN FOR PART #316      '
      EXEC SQL OPEN S4
      STMT = 'FETCH FOR PART #316      '
      EXEC SQL FETCH S4
1      INTO :PARTNO, :DESCR, :QONHND, :PRICE, :NAME, :ADR
      STMT = 'CLOSE FOR PART #316      '
      EXEC SQL CLOSE S4
      STMT = 'UPDATE QUOTATIONS      '
      EXEC SQL UPDATE QUOTATIONS SET QONORDER = 700
1      WHERE PARTNO = :PARTNO AND
2            QONORDER = 0 AND
3            DELIVERY_TIME = :TIME
      CONTINUE
*****
*      PRINT ON SAMPLE ORDER      *
*****
      TPRCE2 = 700 * PRICE
65     FORMAT (' ', TR2, A4, TR6, A10, I11, TR6, A15, TR1, A35, TR3, F7.2,
1TR4, F9.2)
      WRITE (6,65) '700', DESCR, QONHND, NAME, ADR, PRICE, TPRCE2
*****
*      DELETE PART #323      *
*****
      STMT = 'DELETE 323 FROM QUOTATIONS'
      EXEC SQL DELETE FROM QUOTATIONS
1      WHERE PARTNO = 323
      STMT = 'DELETE 323 FROM INVENTORY '
      EXEC SQL DELETE FROM INVENTORY
1      WHERE PARTNO = 323
      CONTINUE
*****
*      COMMIT WORK      *
*****
      STMT = 'COMMIT WORK      '
      EXEC SQL COMMIT WORK
      CONTINUE
*****
*      PRINT TABLES      *
*****
75     FORMAT ('1', TR10, A43)
      WRITE (6,75) '*** PRINTOUT OF TABLE INVENTORY UPDATED ***'
      CALL PRINT1
85     FORMAT ('1', TR10, A44)
      WRITE (6,85) '*** PRINTOUT OF TABLE QUOTATIONS UPDATED ***'
      CALL PRINT2
95     FORMAT ('1', TR10, A43)
      WRITE (6,95) '*** PRINTOUT OF TABLE SUPPLIERS UPDATED ***'
      CALL PRINT3
      GOTO 110
100    CALL ERROUT (STMT)
110    PRINT *, ('SAMPLE PROGRAM ARISFTN COMPLETED')
      END
      SUBROUTINE PRINT1
*****
*      PRINT ROUTINE FOR TABLES      *
*****
      CHARACTER*40      STMT
      EXEC SQL INCLUDE SQLCA
*
*****

```

```

*                ESTABLISH HOST VARIABLES                *
*****
EXEC SQL BEGIN DECLARE SECTION
INTEGER          PARTNO*2,QONHND
CHARACTER        DESCR*10
CHARACTER*8      ID,PASS
EXEC SQL END DECLARE SECTION
DATA ID          /'SQLDBA '/
DATA PASS        /'SQLDBAPW'/
*****
*                ERROR HANDLING                        *
*****
EXEC SQL WHENEVER SQLWARNING CONTINUE
EXEC SQL WHENEVER SQLERROR GOTO 100
*****
EXEC SQL DECLARE C1 CURSOR FOR SELECT PARTNO, DESCRIPTION,
1      QONHAND
2      FROM INVENTORY
3      ORDER BY PARTNO
45     FORMAT (1H0,A33,/)
PRINT *,('PARTNO      DESCRIPTION      QONHAND')
EXEC SQL WHENEVER NOT FOUND CONTINUE
STMT = 'OPEN C1      PRINT      '
EXEC SQL OPEN C1
STMT = 'FETCH C1      IN PRINT1      '
50     EXEC SQL FETCH C1
1      INTO :PARTNO, :DESCR, :QONHND
IF (SQLCOD .EQ. 0) THEN
55     FORMAT (' ',I6,TR5,A10,TR1,I11)
WRITE (6,55) PARTNO,DESCR,QONHND
GOTO 50
ELSE
STMT = 'CLOSE C1 IN PRINT1      '
EXEC SQL CLOSE C1
ENDIF
GOTO 110
100    CALL ERROUT (STMT)
110    RETURN
END
SUBROUTINE PRINT2
CHARACTER*40      STMT
EXEC SQL INCLUDE SQLCA
*
*****
*                ESTABLISH HOST VARIABLES                *
*****
EXEC SQL BEGIN DECLARE SECTION
INTEGER          PARTNO*2,TIME*2,QONORD,SUPPNO*2
REAL*8           PRICE
CHARACTER*8      ID,PASS
EXEC SQL END DECLARE SECTION
DATA ID          /'SQLDBA '/
DATA PASS        /'SQLDBAPW'/
*****
*                ERROR HANDLING                        *
*****
EXEC SQL WHENEVER SQLWARNING CONTINUE
EXEC SQL WHENEVER SQLERROR GOTO 100
EXEC SQL WHENEVER NOT FOUND CONTINUE
*****
EXEC SQL DECLARE C2 CURSOR FOR
1      SELECT SUPPNO, PARTNO, PRICE, DELIVERY_TIME, QONORDER
2      FROM QUOTATIONS
3      ORDER BY SUPPNO, PARTNO
PRINT *,('SUPPNO      PARTNO      PRICE      DELVERY TIME      QONORDER
1')

```

```

      STMT = 'OPEN C2   PRINT2           '
      EXEC SQL OPEN C2
      STMT = 'FETCH C2   IN PRINT2       '
50    EXEC SQL FETCH C2
      1      INTO :SUPPNO, :PARTNO, :PRICE, :TIME, :QONORD
      IF (SQLCOD .EQ. 0) THEN
55    FORMAT (' ', I6, TR4, I6, TR3, F7.2, TR10, I6, TR3, I11)
      WRITE (6, 55) SUPPNO, PARTNO, PRICE, TIME, QONORD
      GOTO 50
      ELSE
      STMT = 'CLOSE C2 IN PRINT2       '
      EXEC SQL CLOSE C2
      ENDIF
      GOTO 110
100   CALL ERRORT (STMT)
110   RETURN
      END
      SUBROUTINE PRINT3
      CHARACTER*40   STMT
      EXEC SQL INCLUDE SQLCA
*
*****
*           ESTABLISH HOST VARIABLES           *
*****
      EXEC SQL BEGIN DECLARE SECTION
      INTEGER          SUPPNO*2
      CHARACTER        NAME*15, ADR*35
      CHARACTER*8      ID, PASS
      EXEC SQL END DECLARE SECTION
      DATA ID          /'SQLDBA  '/
      DATA PASS        /'SQLDBAPW'/
*****
*           ERROR HANDLING                     *
*****
      EXEC SQL WHENEVER SQLWARNING CONTINUE
      EXEC SQL WHENEVER SQLERROR GOTO 100
      EXEC SQL WHENEVER NOT FOUND CONTINUE
*****
      EXEC SQL DECLARE C3 CURSOR FOR
      1      SELECT SUPPNO, NAME, ADDRESS
      2      FROM SUPPLIERS
      3      ORDER BY SUPPNO
      PRINT *, ('SUPPNO   NAME                ADDRESS')
      STMT = 'OPEN C3   PRINT3           '
      EXEC SQL OPEN C3
      STMT = 'FETCH C3   IN PRINT3       '
50    EXEC SQL FETCH C3
      1      INTO :SUPPNO, :NAME, :ADR
      IF (SQLCOD .EQ. 0) THEN
55    FORMAT (' ', I6, TR4, A15, TR4, A35)
      WRITE (6, 55) SUPPNO, NAME, ADR
      GOTO 50
      ELSE
      STMT = 'CLOSE C3 IN PRINT3       '
      EXEC SQL CLOSE C3
      ENDIF
      GOTO 110
100   CALL ERRORT (STMT)
110   RETURN
      END
      SUBROUTINE ERRORT (STMT)
      EXEC SQL INCLUDE SQLCA
*
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*40   STMT
      INTEGER*4      I
      EXEC SQL END DECLARE SECTION

```

```

PRINT*,'UNEXPECTED SQL ERROR RETURNED')
PRINT*,'FAILING SQL STATEMENT IS  ',STMT
PRINT*,SQLCOD
PRINT*,(SQLERR(I), I=1, 6)
PRINT*,SQLERP
PRINT*,(SQLWRN(I), I=0, 11)
PRINT*,SQLTXL
PRINT*,SQLTXT
*****
*          IGNORE ERRORS DURING ROLLBACK TO AVOID ERROR ROUTINE LOOP  *
*****
EXEC SQL WHENEVER SQLERROR CONTINUE
EXEC SQL ROLLBACK WORK
RETURN
END

```

Rules for Using SQL in FORTRAN

This section lists, for your reference, the rules for embedding SQL statements within a FORTRAN program.

The FORTRAN SQL preprocessor supports programs written for the VS FORTRAN compiler with the LANGLVL (77) option specified. Only FIXED-FORM source statements are supported.

If FORTRAN labels are placed on SQL declarative statements (BEGIN/END DECLARE SECTION, WHENEVER, and INCLUDE), the label will be removed and an information message given.

The FORTRAN preprocessor supports a maximum of 255 program units per input source file (254 subprograms in addition to the main program).

Figure 47 summarizes the SQL statements that are supported by the FORTRAN preprocessor.

Statement	Comments
INCLUDE SQLCA Extended INCLUDE WHENEVER BEGIN/END DECLARE SECTION	No STOP support.
CREATE/DROP TABLE CREATE/DROP VIEW CREATE/DROP INDEX CREATE/DROP SYNONYM DROP PROGRAM ACQUIRE/DROP DBSPACE ALTER TABLE/DBSPACE GRANT/REVOKE LOCK TABLE/DBSPACE COMMENT ON	
INSERT UPDATE DELETE	
SELECT	On DECLARE CURSOR only.
DECLARE CURSOR OPEN CURSOR FETCH CURSOR CLOSE CURSOR	DECLARE CURSOR does not support the dynamic form of the command (DECLARE CURSOR FOR statement-name). Likewise the dynamic form of OPEN is not supported.

Figure 47 (Part 1 of 2). SQL Statements Supported in FORTRAN

Statement	Comments
COMMIT WORK [RELEASE] ROLLBACK WORK [RELEASE]	

Figure 47 (Part 2 of 2). SQL Statements Supported in FORTRAN

Placement of Data Statements

FORTRAN releases prior to 3.0 do not restrict the placement of data statements. Release 3.0 and later require that data statements follow explicit declare statements. Thus in a program containing an SQL DECLARE section, all data statements should follow the END DECLARE SECTION.

Long Character Strings

FORTRAN releases prior to 3.0 do not support SQL character strings longer than 254. To process FORTRAN Release 3.0 character strings that are longer than 254:

1. Declare INTEGER*2 to contain the length of the string
2. Declare CHARACTER*(length) string of data
3. Declare a CHARACTER*(2 + length of string)
4. Declare a COMMON block containing (1) and (2) above
5. Use EQUIVALENCE statement (name of (1) above, name of (2) above)
6. Specify a DATA BLOCK subroutine to initialize (1) and (2) above.

For example:

```

INTEGER*2      STRNGL
CHARACTER*498  STRING
CHARACTER*500  STRNGW
COMMON /SDATA/ STRNGL,STRING
EQUIVALENCE (STRNGW,STRNGL)
.
.
.
(References to string by SQL use (STRNGW)
.
.
BLOCK DATA
COMMON /SDATA/ STRNGL,STRING
INTEGER STRNGL*2 / 498 /
CHARACTER STRING*498
DATA STRING /'.....' /
END

```

Note: When the character string is fetched in the FORTRAN program by SQL, the first two bytes of the string will contain the length of the character string.

Placement and Continuation of SQL Statements

All SQL statements must be placed in columns 7 to 72. Columns 73 to 80 may contain sequence numbers and information; columns 1 to 5 may also contain statement numbers.

The rules for continuation of SQL keywords from one line to the next are the same as the FORTRAN rules for the continuation of words and constants. However, an SQL statement may use up to 124 continuation lines in addition to the first line (for a total of 125). (Note that the maximum length of an SQL statement is 8192 characters.)

Declaring Host Variables

Host variables must be explicitly declared to be used in SQL statements. The following example shows an SQL declare section for a FORTRAN program:

```
EXEC SQL BEGIN DECLARE SECTION  (at beginning of section)
      •
(Data description entries for host variables)
      •
EXEC SQL END DECLARE SECTION    (at end of section)
```

Place the data description entries for all the host variables within the SQL declare sections. You may use the variables appearing in these SQL declare sections in regular FORTRAN statements as well as in SQL statements.

A host variable declared within the SQL DECLARE SECTION may not be continued. That is, the host variable declaration must appear on a single line to be recognized by the preprocessor.

You can also place data description entries for non-host variables in the SQL declare section. That is, the FORTRAN preprocessor ignores data description entries within the SQL declare section that it does not recognize as valid host variable declarations. No error message is generated; instead, the statement is left for the FORTRAN compiler to process. Thus it is possible, but not recommended, to place all data description entries within an SQL declare section.

The rules for declaring variables within SQL declare sections are:

- Host variables must be valid FORTRAN variable names. That is, they can be from one to six characters long, the first character must be a letter (A-Z) or \$, and the remaining characters must be letters (A-Z), numbers (0-9), or \$. Note that many examples in this manual have names that are too large for FORTRAN.
- Variables named in the SQL declare sections must have data descriptions like those in Figure 49 on page 466.
- Variables cannot be any of the following:
 - Vector or array declarations,

- A constant defined by a PARAMETER statement,
 - Any declarations that use expressions to define the length of the variables, or
 - Character variables declared with an undefined length, such as CHARACTER*(*).
- You should not give any variable a name beginning with SQL or SQ, since these names are reserved for SQL/DS use.

A host variable must be declared earlier in the program than the first use of the variable in an SQL statement.

Embedding SQL Statements

You must precede each SQL statement in your program with EXEC SQL. No delimiter should be used at the end of each statement.

FORTTRAN source statements and SQL statements cannot be contained on the same line or within the same continued statement, except when an SQL statement is used as the imperative statement of a logical IF. Also, only one SQL statement can be contained in a single line, or within the same continued statement.

Using Host Variables in SQL Statements

When you place host variables within an SQL statement, you must precede each such variable with a colon (:). The colon distinguishes the host variables from the SQL identifiers (such as PARTNO). When the same variable is used outside of an SQL statement, do not use a colon.

A host variable can represent a data value, but not an SQL identifier. For example, you cannot assign a character constant, such as 'MUSICIANS', to a host variable, and then use that host variable in a CREATE TABLE statement to represent the table name. This pseudo-code sequence is invalid:

```
TT = 'MUSICIANS'  
CREATE TABLE :TT (NAME ...
```

Incorrect

Using Double-Byte Character Set (DBCS) Constants

The FORTRAN preprocessor does not support either DBCS constants or host program variables. Hexadecimal constants, however, can be used for DBCS data representation in SQL statements.

If the DBCS option is set to YES, embedded SQL statements can contain character string constants and identifiers with DBCS characters enclosed by so and si.

Using the INCLUDE Command

To include the external secondary input, specify:

```
EXEC SQL INCLUDE text-name
```

at the point in the source code where the secondary input is to be included. Text-name is the filename of a CMS file with a "FORTCOPY" filetype and located on a CMS minidisk accessed by the user.

FORTRAN Data Conversion Notes

Host variables must be type-compatible with the columns with which they are to be used.

A column of type INTEGER, SMALLINT, or DECIMAL is compatible with a FORTRAN variable of INTEGER, INTEGER*2, or INTEGER*4. Of course, an overflow condition may occur if, for example, an INTEGER data item is retrieved into an INTEGER variable, and its current value is too large to fit.

Fixed-length and varying-length character data (CHAR, VARCHAR) are also considered compatible. SQL/DS automatically converts a varying-length string, and vice-versa, when necessary. If a varying-length string is converted to a fixed-length string, it is truncated or padded on the right with blanks to the correct length. SQL/DS also truncates or pads with blanks if a fixed-length string is assigned to another fixed-length string of a different size (for example, a variable of CHARACTER*12 is stored in a column of type CHAR(18)).

Refer to "Data Conversion" on page 76 for a data conversion summary.

SQL Error Handling

There are two ways to declare the return code structure (called SQLCA):

1. You may write:

```
EXEC SQL INCLUDE SQLCA
```

in your source program. The SQL/DS preprocessor replaces this with the declaration of the SQLCA structure.

2. You may declare the SQLCA structure directly as shown in Figure 48.

```

INTEGER*4      SQLCOD,
*              SQLERR(6),
*              SQLTXL*2
COMMON /SQLCA1/ SQLCOD, SQLERR, SQLTXL

CHARACTER      SQLERP*8,
*              SQLWRN(0:10),
*              SQLTXT*70
COMMON         SQLERP, SQLWRN, SQLTXT

```

Figure 48. SQLCA Structure (in FORTRAN)

The SQLCA must not be declared within the SQL declare section. The meanings of the fields within the SQLCA are discussed under “Error Handling” on page 202.

Dynamic SQL Statements in FORTRAN

Dynamically defined SQL statements are not supported in FORTRAN.

Data Types

Description	SQL/DS Keyword	Equivalent FORTRAN Declaration
A binary integer of 31 bits, plus sign.	INTEGER	INTEGER INTEGER*4
A binary integer of 15 bits, plus sign.	SMALLINT	INTEGER*2
A packed decimal number, precision m, scale n ($1 \leq m \leq 15$ and $0 \leq n \leq m$). In storage the number occupies an even number of bytes up to a maximum of 8 bytes. Precision is the total number of digits. Scale is the number of those digits that are to the right of the decimal point.	DECIMAL(m[,n])	Not supported. FORTRAN INTEGER, REAL, and DOUBLE PRECISION host variables are supported for conversion to and from DECIMAL columns.
A double-precision (8- byte) floating point number, in standard System/370 floating point format.	FLOAT	REAL REAL*4 REAL*8 REAL*16 DOUBLE PRECISION

Figure 49 (Part 1 of 2). SQL/DS Data Types for FORTRAN

Description	SQL/DS Keyword	Equivalent FORTRAN Declaration
A fixed-length character string of length n where n <= 254.	CHAR(n)	CHARACTER CHARACTER*n
A varying-length character string of maximum length n, where n <= 254.	VARCHAR(n)	Not supported. FORTRAN character (fixed length) host variables are supported for conversion to and from VARCHAR and LONG VARCHAR columns.
A varying-length character string of maximum length 32765 bytes (two bytes less than the SQL/DS maximum, because of the length field). (Character strings >= 255 are not supported in FORTRAN releases prior to Release 1.3.)	LONG VARCHAR	Not supported. FORTRAN character (fixed length) host variables are supported for conversion to and from VARCHAR and LONG VARCHAR columns.

Figure 49 (Part 2 of 2). SQL/DS Data Types for FORTRAN

Notes:

1. An * length specification can also be used to override a length specification associated with the initial keyword. The following are examples:

Specification	Valid	Invalid (ignored)
INTEGER VAR001,VAR002(2)	VAR001 4 bytes	VAR002
INTEGER*2 VAR001*4,VAR002	VAR001 4 bytes VAR002 2 bytes	
INTEGER*4 VAR001*2/10/,VAR002*4	VAR001 2 bytes VAR002 4 bytes	
INTEGER*5 VAR001*2,VAR002*4		VAR001,VAR002
REAL VAR001*8,VAR002	VAR001	VAR002
REAL*8 VAR001,VAR002*4,VAR003	VAR001,VAR003	VAR002
DOUBLE PRECISION VAR001,VAR002*4	VAR001	VAR002
REAL*8 VAR001(10,10)*4,VAR002	VAR002	VAR001
REAL*16 VAR001,VAR002*4,VAR003*8	VAR003	VAR001,VAR002
CHARACTER VAR1,VAR2*80	VAR1 1 byte VAR2 80 bytes	
CHARACTER*10 VAR1,VAR2*80	VAR1 10 bytes VAR2 80 bytes	
CHARACTER*500 VAR1(5),VAR2*1	VAR2 1 byte	VAR1

2. Double-Byte Character Set (DBCS) data types are not supported by the SQL/DS FORTRAN preprocessor.



Index

A

- access modules
 - as affected by DROP SYNONYM 246
 - automatic regeneration of 46, 66
 - dropping 73, 290
 - general description of 44
 - having a reserve-word name 73
 - index considerations for 244, 274
 - invalidated by DROP DBSPACE 230, 288
 - invalidated by DROP INDEX 245, 289
 - invalidated by DROP TABLE 242
 - invalidated by DROP VIEW 145
 - invalidated by REVOKE 66
- access path
 - See access modules
- access, concurrent 232
- accessing tables owned by other users 18
- ACQUIRE DBSPACE statement 227, 263
- active set of a cursor 19
- adding columns to tables 241, 267
- adding rows to tables 34, 305
- addition in SQL expressions 30
- additional predicates 104
- ALL keyword
 - as used in subqueries 122
 - within a SELECT clause 16
 - within built-in functions 32
- ALL PRIVILEGES keyword 63, 68
- ALLOCATE statement of PL/I 382
- ALTER DBSPACE statement 231, 265
- ALTER privilege
 - granting the 63
 - revoking the 67, 317
- ALTER TABLE statement 241, 267
- altering DBSPACE characteristics 231, 265
- altering tables 241, 267
- alternative for parameterized statements 164
- analyzing SQL/DS statements 209, 297
- AND logical operator 27
- ANY keyword 122
- APOST preprocessor parameter 191
- apostrophes
 - as used in constants 29
 - usage restriction for DECLARE 22
 - usage restriction for PREPARE 172
 - use considerations in COBOL
- application body 9, 92
- application programs, sample 95
- application prolog 86
- applications, CMS 93
- ARISAMDB
 - general description of 95
- ARISASMC
 - general description of 95
 - source code for 426, 427
- ARISCBLC
 - general description of 96
 - source code for 397, 399
- ARISFTN 453
 - general description of 96
 - sample code for 454
 - source code for 453
- ARISPLIC
 - general description of 96
 - sample code for 368
 - source code for 367
- arithmetic operators 30
- ASM preprocessor parameter 189
- Assembler
 - acquiring the SQLDSECT area 423
 - data types 446
 - declaring host variables
 - example 89
 - declaring host variables for 441
 - declaring the SQLCA for 444
 - declaring the SQLDA for 445
 - embedding SQL statements
 - example 90
 - embedding SQL statements within 443, 464
 - sample program 95
 - sample programs 425
- Assembler considerations 423
- author, definition of 58
- authority
 - granting to others 62, 300
 - revoking from others 66
- authority, hierarchy of 61
- authorization
 - granting to others 62, 300
 - overview of 57
 - revoking from others 66
- authorization checking for dynamically defined statements 175, 295
- automatic blocking override
 - warning flags set because of 205
- automatic revocation of privileges 66
- automatic rollback
 - due to data definition statements 238
 - due to deadlocks 232
 - warning flags set because of 205
- averages via a built-in function 31
- AVG built-in function 31

B

back out, definition of 232
backing out changes 234, 321
based structures 151
basic form, definition of 7
BEGIN DECLARE SECTION statement 87, 268
between position of a cursor 20
BETWEEN predicate 104
blanks within identifiers 75
BLOCK preprocessor parameter 192
blocking 254
 override 205
 warning flags set because of 205
blocking override
 warning flags set because of 205
blocks of pages 228
braces, as used in statement formats 259
brackets, as used in statement formats 259
built-in functions
 ALL keyword 32
 applied to numeric columns 33
 as used in grouping 113
 AVG function 31
 computed over the empty set 33
 COUNT function 31
 DISTINCT keyword 32
 general usage rules 32
 MAX function 31
 MIN function 31
 SUM function 31
 with a correlated reference 127

C

catalog tables 78, 215
catalog tables, owner of (**SYSTEM**) 79
changing data 37, 324
changing **DBSPACE** characteristics 231, 265
changing the data type of a column 239
CHAR data type
 for Assembler 446
 for **COBOL** 417
 for **FORTRAN** 467
 for **PL/I** 383
character data 29
character string constants 29
CHECK preprocessor parameter 191
checking the **SQLCA** 207
clauses, order of 117
CLOSE statement
 format of 26, 269
 general description of 21
CLOSE statement, extended 361
closed state of a cursor 19

CMS

applications 93

COBOL

column 7 414
continuation of **SQL** statements within 410
COPYBOOKS 413
data conversion considerations for 413
data types 417
declaring host variables
 example 88
 declaring host variables for 411
 declaring **SQLCA** for 415
delimiting **SQL** statements within 410
embedding **SQL** statements
 example 90
 placement of **SQL** statements within 410
programs using **DBCS** data 414
QUOTE compiler option consideration for 412
restrictions on dynamic statements 416
sample program 96, 397
using the **INCLUDE** command 413
COBOL considerations 397
COBOL preprocessor parameter 189
codes, data, as used by **SQLTYPE** 169
colon
 as used in indicator variables 146
column name, maximum length of 365
column names, join considerations for 108
columns
 naming conventions 74
columns, virtual 141, 279
combining queries 132
commas, as used in statement formats 259
COMMENT statement
 of **SQL** 247
 of **SQL**, format 1 270
 of **SQL**, format 2 271
COMMIT WORK statement
 format of 233, 272
committing changes to tables 233, 272
common column names 108
comparison operator 27
compatibility of data types 18
compiling your program 196
completion code
 See **SQLCODE**
concatenation symbol for **SQL/DS** 78
concatenation within **EXECUTE IMMEDIATE** 149
concurrent access 232
concurrent users, maximum number of 366
CONNECT authority
 description of 57
 granting 65, 303
CONNECT considerations 186
CONNECT statement
 format of 273
 in application programs 91
connecting to **SQL/DS**
 in application programs 91

- connecting users without passwords 60
- consistency of data 72
- constants 28
 - as used in search conditions 28, 31
 - as used in UNIONS 134
 - data types of 28, 134
 - within select-list expressions 16, 134
- contention for resources 232
- continuation of SQL statements
 - within Assembler 443
 - within COBOL 410
 - within FORTRAN 463
 - within PL/I 375
- CONTINUE keyword of WHENEVER 207, 330
- conversion of data
 - See data conversion
- conversion of floating point to integer 18
- COPYBOOKs, COBOL 413
- correlated function 127
- correlated subqueries 124
- correlated subqueries using joins 128
- correlation 124
- correlation table 126
- correlation variable 125
- cost estimate for an SQL statement 204
- COUNT built-in function 31
- COUNT(*) built-in function 32
- COUNT(*) within a grouping query 114
- counting via a built-in function 31
- CREATE INDEX statement 242, 274
- CREATE PROGRAM statement 348
- CREATE SYNONYM statement 245, 276
- CREATE TABLE statement 238, 277
- CREATE VIEW statement 140, 279
- creating indexes 242, 274
- creating objects via dynamically defined statements 175
- creating synonyms 245, 276
- creating tables 238, 277
- creating views 140, 279
- creator names
 - naming conventions 74
- creator, definition of 58
- current row of a cursor 19
- cursor management 19
- cursor name
 - maximum length of 365
 - syntax rules for 22
- cursor stability locking 251
- cursors
 - active set of 19
 - as affected by COMMIT WORK 21
 - as affected by ROLLBACK WORK 21
 - between state 20, 138
 - closed state of 19
 - closing 26, 269
 - current row of 19
 - declaring 22, 179, 281, 282
 - deleting rows via 138, 285
 - fetching 23, 181, 299

- for dynamically defined queries 157
- general description of 19
- inserting 25, 181, 316
- maximum number of 366
- naming conventions 74
- open state of 19
- opening 23, 180, 313
- operations on 20
- ordering results of 134
- requirements for deletion via 138, 285
- requirements for updates via 139
- scope of 135
- updating rows via 139, 326

D

- Data Base Administrator (DBA)
 - description of 57
- data base machine SQL/DS 184
- Data Base Storage System (DBSS) code 203
- data codes used by SQLTYPE 169
- data consistency 72
- data control 70
- data conversion
 - as done in unions 133
 - in COBOL 413
 - in dynamically defined statements 165
 - in INSERT statements 35, 137
 - in join conditions 107
 - in PL/I 378
 - in UPDATE statements 38
 - summary of 76
- data definition 74, 237
- data dictionary
 - See catalog tables
- data manipulation
 - DELETE statement 284
 - INSERT statement 34, 305
 - UPDATE statement 37, 324
- data types 134
 - as returned by DESCRIBE 169
 - for Assembler 446
 - for COBOL 417
 - for FORTRAN 466
 - for PL/I 383
 - introduction to 75, 87
- data, inconsistent 72
- data, virtual 141
- DBA (Data Base Administrator)
 - description of 57
- DBA authority
 - general description of 61
 - granting 65, 303
- DBCS and FORTRAN 464
- DBCS constants 99
- DBCS data
 - using in Assembler 444

- using in COBOL 414
- using in FORTRAN 464
- using in PL/I 378
- DBCS data types
 - for COBOL 417
 - for PL/I 383
- DBEXTENTs, description of 70
- DBNAME initialization parameters 187
- DBNAME preprocessor parameter 195
- DBSPACE
 - acquiring 227, 263
 - altering the free space of 231, 265
 - altering the lock size of 231, 265
 - creation of 70
 - definition of 70
 - dropping 230, 288
 - maximum number of tables within 366
 - naming conventions for 74
 - owner of 70
 - specifying the placement of tables within 239
- DBSPACE lock size 231, 266
- DBSPACES
 - naming conventions 74
- DBSS (Data Base Storage System) return code 203
- DCSSID initialization parameters 187
- DCSSID preprocessor parameter 195
- deadlocks
 - description of 232
 - warning flags set because of 205
- DECIMAL data type
 - for Assembler 446
 - for COBOL 417
 - for FORTRAN 466
 - for PL/I 383
- DECIMAL values, range of 365
- declarative SQL statements 92
- DECLARE CURSOR statement
 - for coded inserts 282
 - for coded queries 22, 281
 - for dynamically defined statements 179
- DECLARE CURSOR statement, extended 356
- declare section
 - beginning 268
 - ending 294
- declaring host variables
 - in Assembler 441
 - in COBOL 411
 - in FORTRAN 463
 - in PL/I 376
- declaring static external variables
 - in PL/I 376
- declaring the return code structure
 - for Assembler 444
 - for COBOL 415
 - for PL/I 380
- defining indexes 242, 274
- defining synonyms 245, 276
- defining tables 238, 277
- defining views on views 141
- DELETE privilege
 - granting the 63
 - revoking the 67, 317
- DELETE statement
 - as used in a cursor 20, 135, 138, 285
 - error considerations for 37
 - Format 1 of 36, 284
 - Format 2 of 138, 285
 - meaning of SQLCODE 100 for 37, 284
 - SQLERRD(3) consideration 37
 - warning flags set because of 36, 204, 284
- deleting indexes 245, 289
- deleting rows of tables 284
- deleting synonyms 246, 291
- deleting tables 241, 292
- delimiting SQL statements
 - for dynamic execution 149
 - within Assembler 443
 - within COBOL 410
 - within PL/I 375
- DESCRIBE parameter
 - of extended CREATE PROGRAM statement 349
- DESCRIBE statement
 - format of 177, 286
 - usage techniques for 151
- DESCRIBE statement, extended 352
- descriptor 151
 - See also SQLDA
- destroying access modules 73, 290
- destroying programs 73, 290
- destroying tables 241, 292
- determining the number of rows processed 203
- dictionary, data
 - See catalog tables
- DISTINCT keyword
 - as used within Format 2 INSERT 136
 - within a HAVING clause 116
 - within a SELECT clause 16
 - within built-in functions 32
- division in SQL expressions 30
- double quotes
 - as used in identifiers 75
 - restrictions 75
 - usage restriction for DECLARE 22
 - usage restriction for PREPARE 172
- Double-Byte Character Set
 - See DBCS
- DROP DBSPACE statement 230, 288
- DROP INDEX statement 245, 289
- DROP PROGRAM statement 73, 290
- DROP STATEMENT statement 362
- DROP SYNONYM statement 246, 291
- DROP TABLE statement 241, 292
- DROP VIEW statement 145, 293
- dropping a column from a table 239
- dropping access modules 73, 290
- dropping indexes 245, 289
- dropping programs 73, 290
- dropping rows of tables 36, 284

- dropping synonyms 246, 291
- dropping tables 241, 292
- dropping views 145, 293
- dropping views currently in use 146
- DSECTs used by SQL/DS 423
- duplicates
 - eliminated via unions 133
 - within Format 2 INSERTs 136
- duration of locks 232
- dynamic data conversion 165
- dynamic SQL statements in Assembler 445
- dynamic SQL statements in COBOL 416
- dynamic SQL statements in FORTRAN 466
- dynamic SQL statements in PL/I 381
- dynamic statements
 - comparison with extended dynamic statements 335
- dynamic statements, extended
 - comparison with dynamic statements 335
 - description 333
 - introduction 332
 - logical unit of work considerations 344
 - relationship between 334
- dynamically defined queries 151
- dynamically defined statements
 - creating new objects via 175
 - data conversion for 165
 - estimating cost of 204
 - processing for
 - non-queries 148
 - queries 151
 - SQLDA use in 167

E

- eliminating access modules 73, 290
- eliminating duplicates
 - via unions 133
- eliminating programs 73, 290
- eliminating rows of tables 36, 284
- ellipses, as used in statement formats 260
- empty set
 - built-in functions computed over 33
- END DECLARE SECTION statement 87, 294
- equal sign 28
- erasing indexes 245, 289
- erasing rows of tables 284
- erasing synonyms 246, 291
- erasing tables 241, 292
- error code
 - See SQLCODE
- error conditions 206, 329
- error considerations
 - for Format 1 DELETE 37
 - for Format 1 UPDATE 38
 - for Format 2 INSERT 137, 307
- error handling
 - in application programs 202

- introduction to 51
- error recovery
 - See error handling
- errors, fatal 203, 207
- evaluating predicates, rules for 103
- examining the SQLCA 207
- example tables 9
- exclusive lock, description of 232, 235, 312
- EXECs
 - for sample programs 96
 - SQLINIT 186
 - SQLPREP 187
 - SQLSTART 199
- executable functions
 - See built-in functions
- executable subroutines
 - See built-in functions
- EXECUTE IMMEDIATE statement
 - format of 176, 296
 - usage techniques for 148
- EXECUTE statement
 - format of 175, 295
 - usage techniques for 149
- EXECUTE statement, extended 354
- executing applications
 - in multiple user mode 198
 - in single user mode 199
- executing dynamically defined statements
 - See dynamically defined statements
- execution of SQL/DS requests 45
- execution performance
 - monitoring 209
- EXISTS predicate 131
- expanding tables 241, 267
- EXPLAIN command 209
- EXPLAIN statement 297
- explain-spec, definition 209, 297
- explanation tables 209, 297
- expressing equality 28
- expressing inequality 28
- EXPRESSION m, meaning of, in SQL.NAME 171
- expressions
 - as used in search conditions 30
 - constants used within 31
 - description of 30
 - host variables within 30
- extended CLOSE statement 361
- extended DECLARE CURSOR statement 356
- extended DESCRIBE statement 352
- extended dynamic statements
 - Assembler example 336
 - comparison with dynamic statements 335
 - description 333, 348
 - introduction 332
 - logical unit of work considerations 344
 - relationship between 334
- extended EXECUTE statement 354
- extended FETCH statement 359
- extended OPEN statement 358

extended PREPARE statement 350
extended PUT statement 360
external source files
 including 255
eye-catcher fields
 of the SQLCA 203
 of the SQLDA 168

F

fatal errors 203, 207
fetch and insert blocking 254
FETCH statement
 for dynamically defined queries 181
 format of 23, 299
 general description of 20, 23, 299
FETCH statement, extended 359
FLOAT data type
 for Assembler 446
 for COBOL 417
 for FORTRAN 466
 for PL/I 383
FLOAT values, range of 365
FOR UPDATE clause 135
format of SQL statements 259
Format 1 INSERT 34, 305
 See also INSERT statement
Format 1 UPDATE 37, 324
 See also UPDATE statement
Format 2 DELETE 138, 285
 See also DELETE statement
Format 2 INSERT
 See also INSERT statement
 basic uses of 136, 307
 determining the number of rows inserted by 137
 error considerations for 137, 307
 order of rows added by 137
Format 2 UPDATE 139, 326
 See also UPDATE statement
FORTRAN
 continuation of SQL statements within 463
 data types 466
 declaring host variables
 example 89
 declaring host variables for 463
 embedding SQL statements
 example 91
 long character strings 462
 placement of SQL statements within 463
 sample program 96, 453
 using the INCLUDE command 465
FORTRAN considerations 453
FORTRAN preprocessor parameter 189
FROM clause
 as used in joins 107
 correlation variable within 125
 general description of 18

 of the PREPARE statement 172, 314
 of the PUT statement 181
fully qualify 78
functions, built-in
 See built-in functions

G

general rules for naming data objects 74
GOTO keyword of WHENEVER 207, 330
GRANT option
 general description of 57, 59
 granting the 63, 64
 revoking the 68, 69
GRANT RUN privilege 214
GRANT statement 62, 300
granting authority to others 62, 300
granting privileges
 on programs 63, 302
 on tables and views 62, 300
granting special privileges 65, 303
granting special privileges already owned 66
granting the RUN privilege 63, 302
GRAPHIC data type
 for COBOL 417
 for PL/I 383
GRAPHIC preprocessor parameter 191
greater than or equal to sign 28
greater than sign 28
GROUP BY clause
 correlated subquery considerations for 127
 general description of 113
 subqueries within 123
grouping 113

H

HAVING clause
 correlated subqueries within 127
 general description of 116
 subqueries within 121, 123
header pages 228, 264
hexadecimal constants within expressions 100
hierarchy of authority 61
host language, definition of 7
host variables
 as declared in Assembler 441
 as declared in COBOL 411
 as declared in FORTRAN 463
 as declared in PL/I 376
 as used in Assembler 443
 as used in COBOL 412
 as used in FORTRAN 464
 as used in PL/I 378

- as used in search conditions 30
- declaring 268, 294
 - examples 89
- detecting nulls within 146
- detecting truncation of 146
- maximum length of name 366
- maximum number of 366
- naming conventions 74
- pointed to by SQLDATA 170
- restriction on use within CREATE VIEW 142
- restriction on use within GRANT 62
- restriction on use within REVOKE 66
- within dynamically defined statements 173
- within INTO clauses 18
- within statements to be dynamically executed 149
- host-program variables
 - See host variables
- how SQL/DS joins tables 107
- how to interpret SQL format 259
- how to join tables 107
- hyphens in COBOL 411

I

- identifier, definition of 74
- implicit revocation of privileges 66
- IN predicate 105, 123
- INCLUDE command 255
 - using, in ASSEMBLER 443
 - using, in COBOL 413
 - using, in FORTRAN 465
 - using, in PL/I 375
- INCLUDE SQLCA statement
 - in Assembler language 444
 - in COBOL 415
 - in FORTRAN 465
 - in PL/I 380
 - in pseudo code 51, 86
- INCLUDE SQLDA statement
 - in Assembler language 445
 - in PL/I 381
 - in pseudo code 152
- including external source files 255
- including secondary input from VM
 - CMS file 255
- inconsistent data 72
- inconsistent state 51, 72
- index name, maximum length of 365
- INDEX privilege
 - granting the 63
 - revoking the 67, 317
- indexes
 - creating 242, 274
 - dropping 245, 289
 - maximum length of 366
 - maximum number of columns referenced by 366
 - naming conventions 74

- restriction on use with views 140
- indicator variables
 - declaration of 146
 - general description of 146
 - in dynamic data conversion 167
 - meaning of values returned within 146
 - pointed to by SQLIND 170
 - usage restrictions for EXECUTE 175
 - usage restrictions for OPEN cursor 180
 - used to detect nulls 24, 147
 - used to detect truncation 24, 147
 - within the FETCH statement 24, 181
- initialization parameters
 - DBNAME 187
 - DCSSID 187
- initializing the SQLDA 167
- initializing your user machine 186
- input host variables 20, 23, 313
- insert and fetch blocking 254
- INSERT privilege
 - granting the 63
 - revoking the 67, 317
- INSERT statement
 - data conversion within 35, 137
 - eliminating duplicate rows via 136
 - error considerations for 137, 307
 - inserting nulls via 35
 - order of rows added by 35, 305
 - SELECT statement within (Format 2) 136, 307
 - SQLERRD(3) consideration for 137
 - use of DISTINCT keyword within 136
 - using a list of values (Format 1) 34, 305
- inserting nulls into tables 35
- inserting rows
 - See also INSERT statement
 - using a list of values 34, 305
 - using a SELECT statement 136, 307
- installing applications 184
- INTEGER data type
 - for Assembler 446
 - for COBOL 417
 - for FORTRAN 466
 - for PL/I 383
- INTEGER values, range of 365
- intermediate values of computation, limitation on 365
- internal statistics, updating 237, 328
- interpreting SQL format 259
- INTO clause
 - as used in unions 132
 - general description of 17
 - of the DESCRIBE statement 177, 286
 - of the FETCH statement 23, 181, 299
 - usage restriction for dynamically defined queries 173, 315
 - use restriction for subqueries 121
 - within dynamically defined statements 153
- invoking programs that contain unauthorized statements 58
- invoking/the preprocessor 187

isolation level cursor stability 251
 isolation level repeatable read 251
 isolation levels
 cursor stability 251
 mixing 252, 389
 repeatable read 251
 ISOLATION preprocessor parameter 192

J

join conditions
 data conversion performed on 107
 definition of 107
 nulls within 110
 number permitted 112
 join variable 110
 joining a table to itself 110
 joining tables
 See joins
 joins
 data conversion performed within 107
 introduction to 107
 join variable 110
 limits on 112
 nulls within 110
 of a single table (to itself) 110
 ORDER BY considerations of 112
 referring to another user's table 108, 110
 SELECT * 112
 with common column names 108
 within correlated subqueries 128
 without join conditions 107

K

KEEP parameter
 of extended CREATE PROGRAM statement 348
 KEEP preprocessor parameter 190
 keywords, reserved 363

L

LABEL statement
 format 1 309
 format 2 310
 of SQL 249
 less than or equal to sign 28
 less than sign 28

LIKE predicate 106
 limits of SQL/DS 365
 limits on joins 112
 LINECOUNT preprocessor parameter
 VM/SP 191
 literal constants 28
 literals
 as used in UNIONS 134
 data types of 28
 loading your program 197
 LOCK parameter of ACQUIRE DBSPACE 229, 264
 LOCK parameter of ALTER DBSPACE 231, 265
 lock size
 altering 231, 265
 definition of 71
 LOCK statement 235, 312
 locked DBSPACES, modifying 71
 locked DBSPACES, reading from 71
 locking DBSPACES explicitly 235, 312
 locking duration 232
 locking tables explicitly 235, 312
 locking, description of 232
 locks
 exclusive 232
 share 232
 logical operator
 logical units of work
 as used in error handling 51
 automatic locking of 232
 automatic rollback of 232, 238
 CMS considerations for 93, 234, 321
 committing work done during 233, 272
 considerations in using extended dynamic
 statements 344
 definition of 51, 72
 revoking privileges during 66
 rolling back work done during 234, 321
 LOGMODE preprocessor parameter 196
 long character strings in FORTRAN 462
 long fields 236
 LONG VARCHAR
 general usage restrictions 236
 usage restrictions for unions 134
 LONG VARCHAR data type
 for Assembler 446
 for COBOL 417
 for FORTRAN 467
 for PL/I 383
 LONG VARGRAPHIC
 general usage restrictions 236
 usage restrictions for unions 134
 LONG VARGRAPHIC data type
 for COBOL 418
 for PL/I 383
 lowercase letters within identifiers 75

M

- main variable
 - See host variables
- main variable, definition of 146
- manipulating a cursor 20
- MAX built-in function 31
- maximum length of an SQL statement 366
- maximums on joins 112
- maximums, determining via a built-in function 31
- maximums, SQL/DS 365
- meaning of value returned in indicator variables 146
- merging results of queries 132
- MIN built-in function 31
- minimums, determining via a built-in function 31
- minimums, SQL/DS 365
- mixing isolation levels 252, 389
- MODIFY parameter
 - of extended CREATE PROGRAM statement 349
- modifying a locked DBSPACE 71
- modifying tables through a view 143
- monitoring execution performance 209
- multi-partition mode, locking considerations of 232
- multi-row query results 19
- multiple data base operation 184
- multiple row results 19
- multiple user mode
 - description 184
 - executing applications in 198
 - invoking the preprocessors 187
- multiple virtual machine mode 184
- multiple-partition mode, locking considerations of 232
- multiple-row query results 19
- multiplication in SQL expressions 30

N

- names of prepared statements 172
- naming columns 74
- naming data objects 74
- naming DBSPACES 74
- naming indexes 74
- naming tables 74
- negative SQLCODE, meaning of 52, 86
- nesting correlated subqueries 129
- NEW parameter
 - of extended CREATE PROGRAM statement 349
- NHEADER parameter of ACQUIRE DBSPACE 228, 264
- NOBLOCK preprocessor parameter 192
- NOCHECK preprocessor parameter 191
- NODESCRIBE parameter
 - of extended CREATE PROGRAM statement 349
- NOGRAPHIC preprocessor parameter 191
- NOMODIFY parameter
 - of extended CREATE PROGRAM statement 349

- NOPRINT preprocessor parameter 191
- NOPUNCH preprocessor parameter 191
- NOT BETWEEN predicate 104
- not equal sign 28
- NOT EXISTS predicate 131
- not found SQLCODE (100)
 - as set by DELETE 37, 284
 - as set by FETCH 24
 - as set by UPDATE 38, 325
 - detecting via WHENEVER 206, 329
- NOT IN predicate 105, 123
- NOT keyword 27
- NOT LIKE predicate 106
- NOT NULL option of CREATE TABLE 239, 278
- NOT NULL predicate 105
- notes on constructing search conditions 103
- NULL keyword
 - as used in UPDATE 38
- NULL predicate 105
- null values used within indicator variables 146
- nulls
 - in computing built-in functions 32
 - in creating tables 239, 278
 - in new columns 241, 267
 - in search conditions 101
 - inserting into tables 35
 - using indicator variables to detect 146
 - warning flags set because of 204
 - within correlated subqueries 130
 - within grouping queries 114
 - within joins 110
 - within LONG VARCHAR fields 236
 - within subqueries 122
 - within unions 133
 - within UPDATE statements 38
- number of rows processed 203

O

- open state of a cursor 19
- OPEN statement
 - format of 23, 313
 - general description of 20
 - specific description of 23, 313
 - with USING option 180
- OPEN statement, extended 358
- operators
 - arithmetic 30
 - comparison 27
 - logical 27
- OPTIONS(MAIN) clause 375
- OR logical operator 27
- ORDER BY clause
 - as used in joins 112
 - as used in unions 132, 134
 - description of 134
 - maximum number of columns within 366

- use restriction for CREATE VIEW 141
- order of clauses 117
- order of rows added by INSERT 35, 137, 305
- ordering query results 134
- ordering the results of a join 112
- output host variables 20, 23, 299
- overflow 76
- owner of a DBSPACE 70
- owner of catalog tables (SYSTEM) 79

P

- PAGE lock size 231, 266
- PAGES parameter of ACQUIRE DBSPACE 228, 264
- pages, header 228, 264
- parameterized non-query statements 163
- parameterized queries 159
- parameterized statement, definition of 149
- parameterized statements 159, 163, 164
- parameters, question-mark 149, 173
- parameters, specifying user 199
- parentheses, as used in statement formats 260
- PARMID preprocessor parameter 196
- passwords
 - naming conventions 74
- paths, access
 - See access modules
- pattern, definition of 106
- PCTFREE clause
 - of CREATE INDEX 244, 275
- PCTFREE parameter of ACQUIRE DBSPACE 229, 264
- PCTFREE parameter of ALTER DBSPACE 231, 265
- PCTINDEX parameter of ACQUIRE DBSPACE 228, 264
- performance considerations 250
- performance monitoring 209
- period, as used for SQL/DS concatenation 78
- PL/I
 - attributes of variables 377
 - considerations 367
 - continuation of SQL statements within 375
 - data conversion considerations for 378
 - data types 383
 - declaring host variables
 - example 89
 - declaring host variables for 376
 - declaring SQLCA for 380
 - declaring SQLDA for 381
 - declaring static external variables 376
 - delimiting SQL statements within 375
 - dynamic allocation of SQLDA 381
 - embedding SQL statements
 - example 90
 - placement of SQL statements within 375
 - sample program 96, 367
 - using the INCLUDE command 375

- placement of SQL statements
 - in Assembler 443
 - in COBOL 410
 - in FORTRAN 463
 - in PL/I 375
- placement of tables in DBSPACES 239
- PLI preprocessor parameter 189
- positions of a cursor 19
- positive SQLCODE, meaning of 52, 86
- potential deadlocks 232
- precedence rules 27, 30
- predicates
 - BETWEEN 104
 - constants used within 28, 31
 - description of 27
 - host variables within 30
 - IN 105
 - LIKE 106
 - NULL 105
 - rules for evaluating 103
- preface v
- PREPARE statement
 - format of 172, 314
 - usage techniques for 149
- PREPARE statement, extended 350
- PREPARM preprocessor parameter 190
- PREPNAME preprocessor parameter 73, 190, 290
- preprocessing
 - general description of 44
 - in multiple user mode 187
 - in single user mode 187
 - parameters 189
- preprocessing programs with unauthorized statements 58
- preprocessor parameters 250
 - APOST 191
 - ASM 189
 - BLOCK 192
 - CHECK 191
 - COBOL 189
 - DBNAME 195
 - DCSSID 195
 - FORTRAN 189
 - GRAPHIC 191
 - ISOLATION 192, 252
 - KEEP 190
 - LINECOUNT 191
 - LOGMODE 196
 - NOBLOCK 192
 - NOCHECK 191
 - NOGRAPHIC 191
 - NOPRINT 191
 - NOPUNCH 191
 - PARMID 196
 - PLI 189
 - PREPARM 190
 - PREPNAME 73, 190, 290
 - PRINT 191
 - PUNCH 191
 - QUOTE 191, 412

- REVOKE 190
- SYSIN 192
- SYSPRINT 193
- SYSPUNCH 194
- USERID 190
- preprocessors 188
- PRINT preprocessor parameter 191
- privileges
 - automatic revocation of 66
 - definition of 57
 - granting to others 62, 300
 - revoking from others 66
- PRIVILEGES keyword 63, 68
- privileges on programs
 - description of 58
 - granting 63, 302
 - revoking 68, 319
- privileges on tables and views
 - description of 57
 - granting 62, 300
 - revoking 67, 317
- program creator, definition of 58
- program privileges
 - description of 58
 - granting 63, 302
 - revoking 68, 319
- program termination
 - for CMS programs 93
- programs
 - dropping 73, 290
 - having a reserve-word name 73
 - naming conventions 74
 - that drop their own access module 73
- programs using DBCS data
 - in COBOL 414
- programs, sample 95
- prolog 86
- pseudo code, definition of 7
- PUNCH preprocessor parameter 191
- PUT statement
 - for dynamically defined inserts 181
 - format of 25, 316
 - general description of 20, 25, 316
- PUT statement, extended 360
- putting labels on 309, 310
- putting labels on columns 309, 310
- putting labels on tables 309

Q

- qualifiers
 - for column names 108
 - for table names 78
- qualify, fully 78
- queries
 - dynamically defined 151
 - parameterized 159

- querying tables through a view 142
- question-mark parameters 149, 173
- QUOTE preprocessor parameter 191, 412
- quotes
 - as used in constants 29
 - as used in identifiers 75
 - usage restriction for DECLARE 22
 - usage restriction for PREPARE 172

R

- range of values 365
- RDS (Relational Data System) return code 203
- reading from a locked DBSPACE 71
- REFER feature of PL/I 382
- Relational Data System (RDS) return code 203
- relative cost of an SQL statement 204
- RELEASE option
 - of COMMIT WORK 233, 272
 - of ROLLBACK WORK 234, 321
- releasing your connection
 - in CMS applications 233, 234, 272
- releasing your connection to SQL/DS 233, 234
- repeatable read locking 251
- REPLACE parameter
 - of extended CREATE PROGRAM statement 349
- repositioning cursors 20
- reserved words
 - as used for identifiers 75
 - list of 363
- RESOURCE authority
 - granting 65, 303
- Resource Manager return code 203
- restoring data 234, 321
- restrictions
 - involving unions 133
 - on DISTINCT within a SELECT statement 34
 - on the COUNT built-in function 33
- result code
 - See SQLCODE
- retrieving all the fields of a row 16
- return code
 - See SQLCODE
- revocation, automatic, of privileges 66
- REVOKE parameter
 - of extended CREATE PROGRAM statement 348
- REVOKE preprocessor parameter 190
- REVOKE statement 66
- revoking a privilege currently in use 66
- revoking privileges
 - on programs 68, 319
 - on tables and views 67, 317
 - overview of 66
- revoking the GRANT option 68
- roll back, introduction to 51
- ROLLBACK WORK statement 232, 234, 321
- rollback, automatic

- due to data definition statements 238
- due to deadlocks 232
- warning flags set because of 205
- rolling back changes 234, 321
- ROW lock size 231, 266
- rows, maximum length of 365
- rules for evaluating predicates 103
- rules for naming data objects 74
- rules for using SQL in Assembler 441
- rules for using SQL in COBOL 410
- rules for using SQL in FORTRAN 461
- rules for using SQL in PL/I 375
- RUN authority
 - See RUN privilege
- RUN privilege
 - automatic revocation of 66
 - conditions for receipt of 213
 - description of 58
 - granting 63, 302
 - when revoked by a DBA 66
 - with the GRANT option 59
- run-time statements
 - See dynamically defined statements
- running programs that contain unauthorized statements 58

S

- sample application programs 95
- sample program EXECs 96
- sample programs
 - ARISASM 426
 - ARISCBLC 397
 - ARISFTN 453
 - ARISPLIC 367
- sample tables 9
- SCHEDULE authority 60
- scope
 - of a WHENEVER statement 207
 - of cursors 135
- search condition, definition of 27
- search conditions
 - See also WHERE clause
 - AND logical operator 27
 - arithmetic operators within 30
 - comparison operators within 27
 - constants used within 28, 31
 - expressions within 30
 - host variables within 30
 - join conditions within 107
 - NOT keyword 27
 - OR logical operator 27
 - precedence rules of 27, 30
 - predicates within 27
- SELECT *
 - as applies to views 141, 280
 - as used in a join 112

- as used in basic queries 16
- SELECT clause 15
- SELECT privilege
 - granting the 63
 - revoking the 67, 317
- SELECT statement
 - ALL keyword 16, 122
 - ANY keyword 122
 - built-in functions within 31
 - constants within 16
 - DISTINCT keyword 16
 - EXISTS predicate 131
 - INTO clause of 17
 - introduction to 14
 - involving correlation 124
 - involving grouping 113
 - involving joins 107
 - involving subqueries 119
 - involving unions 132
 - NOT EXISTS predicate 131
 - order of clauses within 14, 117
 - ordering results of 134
 - SELECT * form of 16
 - SELECT clause of 15
 - select-list 15
 - WHERE clause of 19
- select-list restrictions 34
- select-list restrictions for GROUP BY 114
- select-list, definition of 15
- select-lists
 - built-in functions within 31
 - constants within 16
 - maximum number of items within 366
- selecting all the fields of a row 16
- selecting the isolation level 252
- sequence of clauses 117
- SET clause 38, 139
 - See also UPDATE statement
- share lock, description of 232, 235, 312
- single quotes
 - as used in constants 29
 - usage restriction for DECLARE 22
 - usage restrictions for PREPARE 172
- single user mode
 - description 184
 - executing applications in 199
 - invoking a program, example 199
 - invoking the preprocessors 187
 - specifying user parameters 199
- single virtual machine mode 184
- single-partition mode, locking considerations of 232
- single-row query results 17
- small-integer-value, definition 209, 297
- SMALLINT data type
 - for Assembler 446
 - for COBOL 417
 - for FORTRAN 466
 - for PL/I 383
- SMALLINT values, range of 365

source value 76
 special characters within identifiers 75
 special privileges
 description of 60
 granting of 65, 303
 revoking of 69, 320
 when revoked by a DBA 66
 special statements
 DROP PROGRAM 73, 290
 UPDATE STATISTICS 237, 328
 SQL format 259
 SQL identifier, definition of 74
 SQL identifiers, maximum length of 365
 SQL reserved words 363
 SQL statements
 embedding in application program
 examples 91
 SQL statements in PL/I subroutines 380
 SQL statements, maximum length of 366
 SQL syntax 259
 sql-command, definition 210, 298
 SQL/DS catalogs 78, 215
 SQL/DS data base machine 184
 SQL/DS data types, introduction to 75, 87
 SQL/DS maximums 365
 SQL/DS minimums 365
 SQL/DS statements
 ACQUIRE DBSPACE 227, 263
 ALTER DBSPACE 231, 265
 ALTER TABLE 241, 267
 BEGIN DECLARE SECTION 87, 268
 CLOSE 26, 269
 COMMENT 247, 270, 271
 COMMIT WORK 233, 272
 CONNECT 91, 273
 CREATE INDEX 242, 274
 CREATE SYNONYM 245, 276
 CREATE TABLE 238, 277
 CREATE VIEW 140, 279
 DECLARE 22, 179
 DECLARE, format 1 281
 DECLARE, format 2 282
 DELETE 284
 DESCRIBE 177, 286
 DROP DBSPACE 230, 288
 DROP INDEX 245, 289
 DROP PROGRAM 73, 290
 DROP SYNONYM 246, 291
 DROP TABLE 241, 292
 DROP VIEW 145, 293
 END DECLARE SECTION 87, 294
 EXECUTE 175, 295
 EXECUTE IMMEDIATE 176, 296
 EXPLAIN 209, 297
 FETCH 23, 181, 299
 GRANT 62, 300
 INCLUDE 255
 INCLUDE SQLCA 51, 86
 INCLUDE SQLDA 152
 INSERT 34, 305
 LABEL 249, 309, 310
 LOCK 235, 312
 OPEN 23, 180, 313
 PREPARE 172, 314
 PUT 25, 181, 316
 REVOKE 66
 ROLLBACK WORK 234, 321
 SELECT
 basic use of 14
 correlation 124
 grouping 113
 joins 107
 subqueries 119
 testing for existence 131
 unions 132
 UPDATE 37, 324
 UPDATE STATISTICS 237, 328
 WHENEVER 206, 329
 SQL/DS statements, analyzing 209, 297
 SQLCA
 Assembler declaration of 444
 COBOL declaration of 415
 description and use of 202
 FORTRAN declaration of 465
 meaning of fields within 202
 PL/I declaration of 380
 testing the 207
 SQLCABC, description of 203
 SQLCAID, description of 203
 SQLCODE 52, 86, 203
 SQLCODE 100 (not found)
 as set by DELETE 37, 284
 as set by FETCH 24
 as set by UPDATE 38, 325
 detecting via WHENEVER 206, 329
 SQLD field of the SQLDA 168
 SQLDA
 as used in DESCRIBE 177, 286
 as used in EXECUTE 176, 295
 Assembler declaration for 445
 PL/I declaration of 381
 summary of 167
 usage techniques for 151
 SQLDABC field of the SQLDA 168
 SQLDAID field of the SQLDA 168
 SQLDATA field of SQLVAR 170
 SQLDAX structure (in PL/I) 381
 SQLDSECT, acquiring 423
 SQLDSIZ variable 423
 SQLERRD(3)
 as set by Format 1 DELETE 37
 as set by Format 1 UPDATE 39
 as set by Format 2 INSERT 137
 SQLERRD, description of 203
 SQLERRM, description of 203
 SQLERROR condition, definition of 206, 329
 SQLERRP, description of 203
 SQLEXT, description of 206

SQLIND field of SQLVAR 170
 SQLINIT EXEC 186
 parameters 187
 SQLLEN field of SQLVAR 169
 SQLN field of the SQLDA 168
 SQLN, when to set 167
 SQLNAME field of SQLVAR 170
 SQLPREP EXEC 187
 format 189
 parameters 189
 SQLSTART EXEC 199
 example 199
 SQLTYPE field of SQLVAR 168
 SQLVAR array of the SQLDA 168
 SQLWARN, description of 204
 SQLWARNA, description of 205
 SQLWARNING condition, definition of 206, 329
 SQLWARN0, description of 204
 SQLWARN1, description of 204
 SQLWARN2, description of 204
 SQLWARN3, description of 204
 SQLWARN4
 as set by DELETE 36, 284
 as set by UPDATE 38, 325
 description of 204
 SQLWARN5, description of 204
 SQLWARN6
 as set by automatic roll back 232
 description of 205
 SQLWARN7, description of 205
 SQLWARN8, description of 205
 SQLWARN9, description of 205
 statement format, how to interpret 259
 statement name, maximum length of 365
 statement names 172
 statement syntax, how to interpret 259
 statements
 naming conventions 74
 statements, parameterized 159, 163, 164
 statistics on tables 237, 328
 STOP keyword of WHENEVER 207, 330
 storage pools
 definition of 70
 non-recoverable 70
 recoverable 70
 specifying the placement of DBSPACEs within 230, 264
 STORPOOL parameter of ACQUIRE DBSPACE 230, 264
 string-spec, syntax rules for 172, 314
 structures, based 151
 subqueries
 ALL keyword 122
 ANY keyword 122
 IN predicate 123
 introduction to 119
 involving unions (restriction) 134
 many values returned by 122
 NOT IN predicate 123

 nulls within 122
 single value returned by 122
 that use correlation 124
 subtraction in SQL expressions 30
 success code
 See SQLCODE
 SUM built-in function 31
 summary of program framework 94
 summing via a built-in function 31
 synonyms
 creating 245, 276
 dropping 246, 291
 naming conventions 74
 syntax of SQL statements 259
 SYSACCESS catalog table 81
 SYSCATALOG catalog table 81
 SYSCHEMSETS catalog table 82
 SYSCOLAUTH catalog table 80
 SYSCOLUMNS catalog table 81
 SYSDBSPACES catalog table 80
 SYSDROP catalog table 82
 SYSIN preprocessor parameter 192
 SYSINDEXES catalog table 81
 SYSOPTIONS catalog table 82
 SYSPRINT preprocessor parameter 193
 SYSPROGAUTH catalog table 80
 SYSPUNCH preprocessor parameter 194
 SYSSYNONYMS catalog table 81
 SYSTABAUTH catalog table 80
 SYSTEM, owner of catalog tables 79
 SYSUSAGE catalog table 81
 SYSUSERAUTH catalog table 79
 SYSUSERLIST view on SYSUSERAUTH 79
 SYSVIEWS catalog table 81

T

table label 110
 table name, maximum length of 365
 tables
 altering 241, 267
 creating 238, 277
 creating indexes for 242, 274
 creating synonyms for 245, 276
 defining labels for 249
 dropping 241, 292
 dropping indexes created on 245, 289
 dropping synonyms created for 246, 291
 entering comments in SQL/DS catalogs for 247, 270, 271
 explanation 209, 297
 inserting nulls into 35
 maximum number of columns within 365
 maximum number of indexes within 365
 naming conventions 74
 placement of 239
 privileges on 57

SQL/DS catalog 78
target value 76
termination
 of CMS applications 93
testing for existence 131
testing the SQLCA 207
totals via a built-in function 31
truncation 76, 204
truth tables 101
truth value 101

U

underscores in COBOL 411
UNION operator
 description of 132
 eliminating duplicates via 133
 ordering results of 132, 134
 usage restrictions involving
 data types 133
 LONG VARCHAR data 134
 nulls 133
 subqueries 134
 views 134
 use restriction for CREATE VIEW 142
unions
 See UNION operator
unique indexes 243
uniquely identifying a table 78
unknown truth value 101
UPDATE privilege
 granting the 63
 revoking the 67, 317
UPDATE statement
 as used in a cursor 21, 135, 139
 data conversion within 38
 error considerations for 38
 Format 1 of 37, 324
 Format 2 of 139, 326
 how values are computed for 38
 meaning of SQLCODE 100 for 38, 325
 nulls within 38
 SQLERRD(3) consideration 39
 warning flags set because of 38, 204, 325
UPDATE STATISTICS statement 237, 328
updating internal statistics 237, 328
use restrictions on COUNT built-in function 33
USER keyword 100
USERID preprocessor parameter 190
userid
 naming conventions 74
users, concurrent, maximum number of 366
USING clause
 of the EXECUTE statement 175, 295

 of the FETCH statement 181
 of the OPEN statement 180
 of the PUT statement 181
using SQL in Assembler 441
using SQL in COBOL 410
using SQL in FORTRAN 461
using SQL in PL/I 375
using the INCLUDE command
 in COBOL 413
 in FORTRAN 465
 in PL/I 375

V

valid lock sizes 231, 266
VARCHAR data type
 for Assembler 446
 for COBOL 417
 for FORTRAN 467
 for PL/I 383
VARGRAPHIC constants within expressions 99
VARGRAPHIC data type
 for COBOL 417
 for PL/I 383
variable, host
 See host variables
variable, indicator
 See indicator variables
variable, main
 See host variables
vertical bar, meaning of 259
views
 CREATE VIEW statement 140, 279
 DROP VIEW statement 145, 293
 general description of 140
 involving unions (restriction) 134
 modifying tables through 143
 naming conventions 74
 privileges on 57
 querying tables through 142
 SYSUSERLIST 79
virtual columns 141, 279
virtual data 141
VM/CMS file
 including secondary input from 255
VM/SP
 CMS applications 93
 compiling your program 196
 CONNECT considerations 186
 executing applications 198
 initializing your user machine 186
 loading your program 197
 preprocessing programs 187

W

warning conditions 204, 206, 329
warning flags 204, 206, 329
when to invoke UPDATE STATISTICS 237
WHENEVER statement 52, 86, 206, 329
WHERE clause
 ALL keyword 122
 ANY keyword 122
 as used within DELETE statements 36, 284
 as used within UPDATE 37, 324
 correlated subquery within 124
 EXISTS predicate 131
 general description of 19
 grouping considerations for 115
 IN predicate 123
 join conditions with 107
 NOT EXISTS predicate 131
 NOT IN predicate 123
 subqueries within 119

WITH GRANT OPTION

 See GRANT option
words, reserved 363
writing clauses in order 117

Z

zero SQLCODE, meaning of 52, 86

Numerics

100, not found SQLCODE
 as set by DELETE 37, 284
 as set by FETCH 24
 as set by UPDATE 38, 325
detecting via WHENEVER 206, 329



Order No. SH24-5068-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

	Yes		No
• Does the publication meet your needs?	<input type="checkbox"/>		<input type="checkbox"/>
• Did you find the material:			
Easy to read and understand?	<input type="checkbox"/>		<input type="checkbox"/>
Organized for convenient use?	<input type="checkbox"/>		<input type="checkbox"/>
Complete?	<input type="checkbox"/>		<input type="checkbox"/>
Well illustrated?	<input type="checkbox"/>		<input type="checkbox"/>
Written for your technical level?	<input type="checkbox"/>		<input type="checkbox"/>
• What is your occupation?	_____		
• How do you use this publication:			
As an introduction to the subject?	<input type="checkbox"/>	As an instructor in class?	<input type="checkbox"/>
For advanced knowledge of the subject?	<input type="checkbox"/>	As a student in class?	<input type="checkbox"/>
To learn about operating procedures?	<input type="checkbox"/>	As a reference manual?	<input type="checkbox"/>

Your comments:

If you would like a reply, please supply your name and address on the reverse side of this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Cut or Fold Along Line

Reader's Comment Form

Fold and Tape

Please Do Not Staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department G60
P. O. Box 6
Endicott, New York 13760

Fold

Fold

If you would like a reply, please print:

Your Name _____
Company Name _____ Department _____
Street Address _____
City _____
State _____ Zip Code _____
IBM Branch Office serving you _____



Order No. SH24-5068-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
 Please use pressure sensitive or other gummed tape to seal this form.

- | | <i>Yes</i> | | <i>No</i> |
|---|--------------------------|----------------------------|--------------------------|
| • Does the publication meet your needs? | <input type="checkbox"/> | | <input type="checkbox"/> |
| • Did you find the material: | | | |
| Easy to read and understand? | <input type="checkbox"/> | | <input type="checkbox"/> |
| Organized for convenient use? | <input type="checkbox"/> | | <input type="checkbox"/> |
| Complete? | <input type="checkbox"/> | | <input type="checkbox"/> |
| Well illustrated? | <input type="checkbox"/> | | <input type="checkbox"/> |
| Written for your technical level? | <input type="checkbox"/> | | <input type="checkbox"/> |
| • What is your occupation? | _____ | | |
| • How do you use this publication: | | | |
| As an introduction to the subject? | <input type="checkbox"/> | As an instructor in class? | <input type="checkbox"/> |
| For advanced knowledge of the subject? | <input type="checkbox"/> | As a student in class? | <input type="checkbox"/> |
| To learn about operating procedures? | <input type="checkbox"/> | As a reference manual? | <input type="checkbox"/> |

Your comments:

If you would like a reply, please supply your name and address on the reverse side of this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

SQL/Data System Application Programming for VM/System Product (File No. S370/4300-50) Printed in U.S.A. SH24-5068-0

Cut or Fold Along Line

Fold and Tape

Please Do Not Staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.



POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department G60
P. O. Box 6
Endicott, New York 13760

Fold

Fold

If you would like a reply, please print:

Your Name _____
Company Name _____ Department _____
Street Address _____
City _____
State _____ Zip Code _____
IBM Branch Office serving you _____



Order No. SH24-5068-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

	Yes		No
• Does the publication meet your needs?	<input type="checkbox"/>		<input type="checkbox"/>
• Did you find the material:			
Easy to read and understand?	<input type="checkbox"/>		<input type="checkbox"/>
Organized for convenient use?	<input type="checkbox"/>		<input type="checkbox"/>
Complete?	<input type="checkbox"/>		<input type="checkbox"/>
Well illustrated?	<input type="checkbox"/>		<input type="checkbox"/>
Written for your technical level?	<input type="checkbox"/>		<input type="checkbox"/>
• What is your occupation?			
• How do you use this publication:			
As an introduction to the subject?	<input type="checkbox"/>	As an instructor in class?	<input type="checkbox"/>
For advanced knowledge of the subject?	<input type="checkbox"/>	As a student in class?	<input type="checkbox"/>
To learn about operating procedures?	<input type="checkbox"/>	As a reference manual?	<input type="checkbox"/>

Your comments:

If you would like a reply, please supply your name and address on the reverse side of this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

SOL/Data System Application Programming for VM/System Product (File No. S370/4300-50) Printed in U.S.A. SH24-5068-0

Cut or Fold Along Line

Fold and Tape

Please Do Not Staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department G60
P. O. Box 6
Endicott, New York 13760

Fold

Fold

If you would like a reply, *please print:*

Your Name _____
Company Name _____ Department _____
Street Address _____
City _____
State _____ Zip Code _____
IBM Branch Office serving you _____



SAMPLE TABLES

INVENTORY	PARTNO	DESCRIPTION	QONHAND
	207	GEAR	75
	209	CAM	50
	221	BOLT	650
	222	BOLT	1250
	231	NUT	700
	232	NUT	1100
	241	WASHER	6000
	285	WHEEL	350
	295	BELT	85

SUPPLIERS	SUPPNO	NAME	ADDRESS
	51	DEFECTO PARTS	16 BUM ST., BROKEN HAND WY
	52	VESUVIUS, INC.	512 ANCIENT BLVD., POMPEII NY
	53	ATLANTIS CO.	8 OCEAN AVE., WASHINGTON DC
	54	TITANIC PARTS	32 LARGE ST., BIGTOWN TX
	57	EAGLE HARDWARE	64 TRANQUILITY PLACE, APOLLO MN
	61	SKY PARTS	128 ORBIT BLVD., SIDNEY AUSTRALIA
	64	KNIGHT LTD.	256 ARTHUR COURT, CAMELOT ENGLAND

QUOTATIONS	SUPPNO	PARTNO	PRICE	DELIVERY_TIME	QONORDER
	51	221	.30	10	50
	51	231	.10	10	0
	53	222	.25	15	0
	53	232	.10	15	200
	53	241	.08	15	0
	54	209	18.00	21	0
	54	221	.10	30	150
	54	231	.04	30	200
	54	241	.02	30	200
	57	285	21.00	14	0
	57	295	8.50	21	24
	61	221	.20	21	0
	61	222	.20	21	200
	61	241	.05	21	0
	64	207	29.00	14	20
	64	209	19.50	7	7

PARTNO -- SMALLINT (NOT NULL)
 DESCRIPTION -- VARCHAR(24)
 QONHAND -- INTEGER
 SUPPNO -- SMALLINT (NOT NULL)
 NAME -- CHAR(15)
 ADDRESS -- VARCHAR(35)
 PRICE -- DECIMAL(5,2)
 DELIVERY_TIME -- SMALLINT
 QONORDER -- INTEGER

SH24-5068-0

IBM

SH24-5068-0

