Application System/400™

SC21-9609-1

Programming: Structured Query Language/400 Programmer's Guide

IBM

Application System/400™

SC21-9609-1

Programming: Structured Query Language/400 Programmer's Guide

Second Edition (September 1989)

This major revision makes obsolete SC21-9609-0.

See "About This Manual" for a summary of major changes to this edition. Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

This edition applies to Release 2 Modification Level 0 of IBM Structured Query Language/400 (SQL/400) Licensed Program (Program 5728-ST1), and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions or technical newsletters.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

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 licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

The numbers at the bottom right of illustrations are publishing control numbers and are not part of the technical content of this manual.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to your IBM-approved remarketer.

This publication could contain technical inaccuracies or typographical errors.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department 245, Rochester, Minnesota, U.S.A. 55901. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Application System/400, AS/400, COBOL/400, Operating System/400, OS/400, RPG/400, C/400 and SQL/400 are trademarks of the International Business Machines Corporation.

400 is a registered trademark of the International Business Machines Corporation.

© Copyright International Business Machines Corporation 1988, 1989. All rights reserved.

About This Manual

This manual explains to programmers and database managers how to use the IBM Structured Query Language/400 (SQL/400) licensed program, how to access data in a database, and how to prepare, run, and test an application program containing SQL statements.

This manual may refer to products that are announced but are not yet available.

This manual contains small programs which are furnished by IBM as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

Who Should Use This Manual

This manual is intended for use by application programmers and database managers who are familiar with COBOL/400, AS/400 PL/I, C/400, or RPG III (part of RPG/400) languages and who understand basic database applications.

What You Should Know

You should know how to use and write application programs for the AS/400 system. You should also know how to program with COBOL/400, AS/400 PL/I, C/400, or RPG III.

Assumptions Relating to Examples of SQL Statements

The examples of SQL statements shown in this manual are based on the sample tables in Appendix B and assume the following:

- They are shown in the interactive SQL environment or written in COBOL. EXEC SQL and END-EXEC are used to delimit an SQL statement in a COBOL program. A description of how to use SQL statements in a COBOL program is provided in Chapter 5.
- Each SQL example is shown on several lines, with each clause of the statement on a separate line.
- SQL reserved words are highlighted.
- SQL naming convention is used.
- The APOST and APOSTSQL precompiler options are assumed (although they are not the default in COBOL). Character string literals within SQL and host language statements are delimited by apostrophes (').
- The complete syntax of the SQL statement is usually not shown in any one example. For the complete description and syntax of any of the statements described in this manual, see the *SQL/400 Reference* manual.

Whenever the examples vary from these assumptions, it is stated.

© Copyright IBM Corp. 1988, 1989

Because this manual is intended for the application programmer, most of the examples are shown as if they were written in an application program. However, many examples can be slightly changed and run interactively by using the interactive SQL (see Chapter 11). The syntax of an SQL statement, when using interactive SQL, differs slightly from the format of the same statement when it is embedded in a program.

How This Manual Is Organized

Chapter 1 introduces you to the Structured Query Language (SQL) by explaining the concepts and objects of SQL. A comparison of the AS/400 system and the SQL naming rules is shown, and the SQL catalog is described.

Chapter 2 describes how to create and work with collections, tables, and views, and how to use catalogs in database design.

Chapter 3 describes coding techniques for using basic SQL statements and clauses, using a cursor, performing complex search conditions, and joining data from more than one table.

Chapter 4 describes common concepts and rules for using SQL with host languages.

Chapter 5 describes how to use SQL statements in COBOL programs.

Chapter 6 describes how to use SQL statements in RPG III programs.

√Chapter 7 describes how to use SQL statements in PL/I programs.

Chapter 8 describes how to use SQL statements in C programs.

Chapter 9 describes how to issue SQL statements that are defined and run at program run time.

Chapter 10 describes how to prepare and run a program containing SQL statements.

Chapter 11 describes how to use interactive SQL to prompt, syntax check, validate, and run SQL statements.

Chapter 12 describes the security plan for protecting SQL data.

Chapter 13 describes how to establish a test environment for and how to debug SQL statements in an application program.

Chapter 14 describes guidelines and techniques for fine-tuning SQL statements in an application program and for working with retrieved data.

Appendix A contains sample tables and the statements for creating them for inserting information into them. The examples used throughout this manual are based on these sample tables.

Appendix B contains a description of SQL return codes.

Appendix C contains sample programs using SQL/400 statements.

This manual also contains a glossary of terms and abbreviations and an index. Use the glossary to find the meaning of unfamiliar terms. Use the index to look up a topic.

How This Manual Has Changed

The following is a list of the major changes or additions that have been made to this manual:

- A new chapter on the C/400 precompiler
- A new parameter, TGTRLS, allowed on CRTSQLRPG and CRTSQLCBL commands
- New parameters, PGMLNG, DECPNT, and SQLSTRDLM, added to STRSQL command
- Prompting for SQL statements
- Ability to save interactive SQL session

Changes since the previous edition of the manual are indicated by a vertical line to the left of the change.

Related Online Information

The following online information is available on the AS/400 system. After pressing the Help key on any menu, you can press the Help key a second time to see an explanation of how the online information works, including the index search function. You can press either the Help key or F1 for help.

Help for Displays

You can press the Help key on any display to see information about the display. There are two types of help available:

Field
Extended

Field help explains the field on which the cursor is positioned when you press the Help key. For example, it describes the choices available for a prompt. If a system message appears at the bottom of the display, position the cursor on the message and press the Help key to see information about the cause of the message and the appropriate action to take.

Extended help explains the purpose of the display. Extended help appears if you press the Help key when the cursor is outside the areas for which field help is available.

To exit the online information, press F3 (Exit). You return to the display on which you pressed the Help key.

Index Search

Index search allows you to specify words or phrases that identify the information that you want to see. To use index search, press the Help key, then press F11 (Search index). You can also use index search by entering the Start Index Search (STRIDXSCH) command on any command line or by selecting option 2 on the User Support menu.

Help for Control Language Commands

To see prompts for parameters for a control language command, type the command, then press F4 (Prompt) instead of the Enter key. To see extended help for the command, type the command and press the Help key.

Online Education

AS/400 online education provides training on a wide variety of topics. To use the online education, press F13 (User support) on any system menu to show the User Support menu. Then select the option to use online education.

Question-and-Answer Function

The question-and-answer (Q & A) function provides answers to questions you may have about using the AS/400 system. To use the Q & A function, press F13 (User support) on any system menu to show the User Support menu. Then select the option to use the question-and-answer function. You can also use the question-and-answer function by entering the Start Question and Answer (STRQST) command on any command line.

Related Printed Information

If you need more information about how to use SQL statements, statement syntax and parameters, see the following manuals:

- *Programming: Structured Query Language/400 Reference*, SC21-9608

  This manual also contains information on the following:

  – Basic syntax of SQL and language elements (such as characters, tokens, and constants)
  – Scalar functions and column functions
  – Subselect, fullselect, and select-statement queries
  – Limits set by the SQL/400 program
  – SQLCA and SQLDA control blocks

- *Systems Application Architecture Common Programming Interface Database Reference*, SC26-4348.

If you need more information about the interactive data definition utility, see the *Utilities: Interactive Data Definition Utility User's Guide*, SC21-9657.

For more information about AS/400 system security, see the *Programming: Security Concepts and Planning*, SC21-8083.

For more information about entering source and syntax checking of host language and SQL statements, see the *Application Development Tools: Source Entry Utility User's Guide and Reference*, SC09-1172.

For more information on AS/400 control language commands and system programming, see the following:

- *Languages: COBOL/400 User's Guide*, SC09-1158
- *Languages: PL/I Reference Summary*, SX09-1051
- *Languages: PL/I User's Guide and Reference*, SC09-1156
- *Languages: RPG/400 User's Guide*, SC09-1161
- *Languages: C/400 User's Guide* (available at a later date)
- *Languages: C/400 Reference Summary* (available at a later date)
- *Programming: Command Reference Summary*, SC21-8076
- *Programming: Control Language Programmer's Guide*, SC21-8077
- *Programming: Control Language Reference*, SBOF-0481

For more information about databases, see the following:

- *Programming: Backup and Recovery Guide*, SC21-8079
- *Programming: Database Guide*, SC21-9659
- *Programming: Data Description Specifications Reference*, SC21-9620

Contents

Chapter 1. Introduction to Structured Query Language/400 1-1
SQL Concepts 1-1
Relational Databases and Terminology 1-2
Types of SQL Statements 1-2
SQL Objects 1-3
Collections 1-3
Data Dictionary 1-3
Journals and Journal Receivers 1-3
Tables, Rows, and Columns 1-3
Views 1-4
Indexes 1-5
Catalogs 1-5
Catalog Views 1-6
SYSCOLUMNS 1-6
SYSINDEXES 1-8
SYSKEYS 1-8
SYSTABLES 1-9
SYSVIEWDEP 1-9
SYSVIEWS 1-10
Compiled Application Program Objects 1-10
User Source File Member 1-10
Temporary Source File Member 1-11
Program 1-11

Chapter 2. Working with SQL Collections, Tables, Views, Indexes, and Catalogs 2-1
Creating an SQL Collection 2-1
Creating and Using a Table 2-1
Creating the Department Table (TDEPT) 2-2
Using the LABEL ON Statement for Defining Table Names and Column Headings 2-2
Inserting Information into a Table 2-2
Getting Information from a Single Table 2-4
Getting Information from More Than One Table 2-5
Changing Information in a Table 2-6
Deleting Information in a Table 2-6
Creating and Using a View 2-7
Creating a View on a Single Table 2-7
Creating a View Combining Data from More Than One Table 2-8
Notes on Using a View 2-9
Working with Indexes 2-10
Using the Catalog in Database Design 2-11
Getting Catalog Information about a Table 2-11
Getting Catalog Information about a Column 2-11
Getting Catalog Information about Indexes 2-12
Getting Catalog Information about Views 2-12
Building a View over Catalog Views 2-12
Using COMMENT ON 2-13
Getting Comments 2-13

Chapter 3. SQL Coding Techniques 3-1
Using Basic SQL Statements and Clauses 3-1
The INSERT Statement 3-1

© Copyright IBM Corp. 1988, 1989

The UPDATE Statement . . . 3-3
The DELETE Statement . . . 3-4
The SELECT INTO Statement . . . 3-5
The WHERE Clause . . . 3-8
The GROUP BY Clause . . . 3-11
The HAVING Clause . . . 3-13
The ORDER BY Clause . . . 3-14
Using the UNION Keyword to Combine Subselects . . . 3-16
Restrictions for the Length and Data Type of Columns . . . 3-18
Specifying UNION ALL . . . 3-19
Using a Cursor . . . 3-20
Example of How to Use a Cursor . . . 3-20
A Unit of Recovery and Open Cursors . . . 3-25
Preventing Duplicate Rows . . . 3-25
Performing Complex Search Conditions . . . 3-26
Keywords for Use in Search Conditions . . . 3-26
Joining Data from More Than One Table . . . 3-29
The WHERE Clause and Joining Tables . . . 3-30
Notes on the Join Technique . . . 3-31
Inserting Multiple Rows into a Table . . . 3-32
Inserting Default Values into Columns . . . 3-33

Chapter 4. Common Concepts and Rules for Using SQL with Host Languages . 4-1
Using Host Variables in SQL Statements . . . 4-1
Assignment Rules . . . 4-2
Indicator Variables . . . 4-4
Handling SQL Error Return Codes . . . 4-6
Handling Exception Conditions with the WHENEVER Statement . . . 4-7

Chapter 5. Using SQL Statements in COBOL Programs . . . 5-1
Application Requirements . . . 5-1
SQL Communication Area (SQLCA) . . . 5-1
Coding Requirements . . . 5-2
Host Structures . . . 5-4
Basic Requirements for Host Variables . . . 5-5
Coding Rules . . . 5-5
Assignment Rules . . . 5-5
Allowable COBOL Declarations . . . 5-6
Indicator Variables . . . 5-8
External Descriptions . . . 5-8
The WHENEVER Statement . . . 5-8
Handling SQL Error Return Codes . . . 5-8

Chapter 6. Using SQL Statements in RPG Programs . . . 6-1
Application Requirements . . . 6-1
SQL Communication Area (SQLCA) . . . 6-1
Coding Requirements . . . 6-2
Host Structures . . . 6-4
Basic Requirements for Host Variables . . . 6-5
Coding Rules . . . 6-5
Assignment Rules . . . 6-5
Allowable RPG Declarations . . . 6-5
Indicator Variables . . . 6-6
External Descriptions . . . 6-6
The WHENEVER Statement . . . 6-7
Handling SQL Error Return Codes . . . 6-7

Chapter 7. Using SQL Statements in PL/I Programs 7-1
Application Requirements 7-1
SQL Communication Area (SQLCA) 7-1
SQL Descriptor Area (SQLDA) 7-2
Coding Requirements 7-2
Using PL/I Host Variables in SQL Statements 7-3
Host Structures 7-3
Basic Requirements for Host Variables 7-4
Coding Rules 7-4
Assignment Rules 7-5
Allowable PL/I Declarations 7-5
Indicator Variables 7-7
Using the %INCLUDE Directive for External File Descriptions 7-8
Element Description 7-8
Structure Definition 7-9
Structure Ending 7-10
%INCLUDE Example 7-11
The WHENEVER Statement 7-11
Handling SQL Error Return Codes 7-11

Chapter 8. Using SQL Statements in C Programs 8-1
Application Requirements 8-1
SQL Communication Area (SQLCA) 8-1
SQL Descriptor Area (SQLDA) 8-2
Coding Requirements 8-3
Using C Host Variables in SQL Statements 8-4
Host Structures 8-4
Basic Requirements of Host Variables 8-5
Coding Rules 8-5
Assignment Rules 8-6
Allowable C Declarations 8-6
Supported Pointer Data Types 8-8
Indicator Variables 8-9
The WHENEVER Statement 8-9
Handling SQL Error Return Codes 8-9

Chapter 9. Dynamic SQL Applications 9-1
Designing and Running a Dynamic SQL Application 9-3
Processing NonSelect-Statements 9-3
Using the PREPARE and EXECUTE Statements 9-4
Processing Select-Statements and Using SQLDA 9-5
Fixed-List Select-Statements 9-5
Varying-List Select-Statements 9-6
The SQL Descriptor Area (SQLDA) 9-7
SQLDA Format 9-7
Example of a Select-Statement for Allocating Storage for SQLDA 9-10
Using a Cursor 9-14
Using Parameter Markers 9-15

Chapter 10. Preparing and Running a Program with SQL Statements 10-1
Basic Processes of the SQL Precompiler 10-1
Input to the Precompiler 10-2
Output from the Precompiler 10-2
Precompiler Commands 10-7
Syntax for the Precompiler Commands 10-7
Precompiler Command Parameters 10-12

Parameter Definitions 10-12
Required Parameter 10-12
Optional Parameters 10-13
Example of the Precompiler Source Command 10-17
Compiling an Application Program 10-17
Binding an Application 10-18
Program References 10-18
Running a Program with Embedded SQL 10-19
OS/400 DDM Considerations 10-19
Override Considerations 10-19
SQL Return Codes 10-19

Chapter 11. Using Interactive SQL 11-1
Overview 11-1
Terminology 11-2
Getting Started 11-3
Functional Description 11-4
Statement Entry 11-4
Session Services 11-6
List Selection Function 11-8
Exit Interactive SQL 11-8
Help 11-9
The Session and Its Functions 11-9
Recovering a Saved or Failed SQL Session 11-9
Messages 11-9
Supported SQL Statements 11-10
Interactive Session Display Flow Diagram 11-10
Tips on Using Interactive SQL 11-14
Using the List Selection Function 11-14
Testing Your SQL Statements Using Interactive SQL 11-17
Entering DBCS Data 11-17
STRSQL Command 11-18
Example 11-20

Chapter 12. SQL Data Protection 12-1
SQL Security 12-1
Authorization ID 12-1
Public Authority 12-1
Views 12-1
SQL Data Integrity 12-2
Concurrency 12-2
Atomic Operations 12-3
Journaling 12-3
Commitment Control 12-4
Save/Restore 12-6
Damage Tolerance 12-6
Index Recovery 12-6

Chapter 13. Testing SQL Statements in Application Programs 13-1
Establishing a Test Environment 13-1
Designing a Test Data Structure 13-1
Debugging Your Program 13-2

Chapter 14. Guidelines and Techniques for Using SQL 14-1
Guidelines for Using SQL Statements 14-1
Effectively Using an SQL Index 14-2

Improving Performance When Selecting Data from Two or More Tables . . 14-5
Improving Performance by Reducing the Number of Opens 14-6
Improving Performance by Using Blocking Considerations 14-7
Improving Performance when Paging Interactively Displayed Data 14-7
Techniques for Solving Some Common Collection Problems 14-8
Paging through Retrieved Data 14-8
Keeping a Copy of the Data 14-8
Retrieving Data a Second Time 14-8
Establishing Position at the End of a Table 14-10
Adding Data to the End of a Table 14-10
Updating Data as It Is Retrieved from a Table 14-10
Updating Data Previously Retrieved 14-11
Changing the Table Definition 14-11

Appendix A. SQL Sample Tables A-1
Creating the Tables A-1
Department Table (TDEPT) A-1
Employee Table (TEMPL) A-2
Project Table (TPROJ) A-2
Employee Project Account Table (TEMPRACT) A-2
Inserting Information into the Tables A-3
TDEPT Table A-3
TEMPL Table A-3
TPROJ Table A-4
TEMPRACT Table A-4
Sample Tables A-5

Appendix B. SQLCODES B-1
SQLCODE Descriptions B-2
Positive SQLCODEs B-2
Negative SQLCODEs B-2

Appendix C. Sample Programs Using SQL/400 Statements C-1
SQL Statements in COBOL Programs C-3
SQL Statements in PL/I Programs C-11
SQL Statements in RPG Programs C-16
SQL Statements in C Programs C-21
Report Produced by Sample Programs C-25

Glossary G-1

Index X-1

Chapter 1. Introduction to Structured Query Language/400

This manual describes the AS/400[1] system implementation of the Structured Query Language/400 (SQL/400)[1]. SQL manages information based on the relational model of data. SQL statements may be embedded in high-level languages or may be run interactively.

SQL consists of statements and clauses that describe what you want to do with the data in a database and under what conditions you want to do it.

SQL Concepts

SQL/400 consists of three main parts:

- SQL run-time support

  This part supplies the parsing of SQL statements and the support to run any SQL statement. SQL/400 interfaces with the existing system functions to use SQL statements. This support is part of the Operating System/400 (OS/400)[1], which allows applications that contain SQL statements to be run on systems where SQL is not installed.

- SQL precompilers

  This part supports precompiling embedded SQL statements in host languages. These languages are supported: COBOL/400[1], AS/400 PL/I, C/400[1], and RPG III (part of RPG/400[1]). The SQL host language precompilers prepare an application program containing SQL statements. The host language compilers then compile the precompiled host source programs. For more information on precompiling, see Chapter 10.

- SQL interactive interface

  This part supplies you with an interactive interface for creating and running SQL statements. For more information on interactive SQL, see Chapter 11.

[1] AS/400, COBOL/400, Operating System/400, OS/400, RPG/400, C/400, and SQL/400 are trademarks of the International Business Machines Corporation.

© Copyright IBM Corp. 1988, 1989

Relational Databases and Terminology

In the relational model of data, all data is perceived as existing in tables. SQL/400 objects are created and maintained as AS/400 system objects. The following table shows the relationship between AS/400 system terms and SQL relational database terms. For more information on database, see the *Database Guide*.

Table 1-1. Relationship of System Terms to SQL Terms

System Terms	SQL Terms
Library. Groups related objects and allows the user to find the objects by name.	**Collection**. Consists of a library, a journal, a journal receiver, a data dictionary, and an SQL catalog. A collection groups related objects and allows the user to find the objects by name.
Physical file. A set of records.	**Table**. A set of columns and rows.
Record. A set of fields.	**Row**. The horizontal part of a table containing a serial set of columns.
Field. One or more characters of related information of one data type.	**Column**. The vertical part of a table of one data type.
Logical file. A subset of fields and records of one or more physical files.	**View**. A subset of columns and rows of one or more tables.

Types of SQL Statements

There are two basic types of SQL statements: data definition statements (DDL) and data manipulation statements (DML). SQL data definition statements can only operate on objects created by SQL in an SQL collection. SQL data manipulation statements can operate on objects created by SQL as well as AS/400 externally described physical files and AS/400 single-format logical files, whether or not they reside in an SQL collection. The IDDU dictionary definition for program-described files will not be referenced. Program-described files will appear as a table with only a single-character column.

The following SQL statements are data definition statements:

COMMENT ON
CREATE COLLECTION
CREATE INDEX
CREATE TABLE
CREATE VIEW
DROP
GRANT
LABEL ON
REVOKE

The following SQL statements are data manipulation statements:

CLOSE
COMMIT
DECLARE CURSOR
DELETE
FETCH
INSERT
LOCK TABLE
OPEN
ROLLBACK
SELECT
UPDATE

SQL Objects

SQL objects used on the AS/400 system are collections, tables, views, indexes, and catalogs. SQL creates and maintains these objects as AS/400 database objects. A brief description of these objects follows.

Collections

A **collection** consists of a library, a journal, a journal receiver, a catalog, and a data dictionary. Tables, views, and other system objects (such as programs) can be created, moved, or restored in an SQL collection. SQL tables, views, and indexes can only be created into an SQL collection. They cannot be created into an AS/400 library.

AS/400 physical files can be placed in a created, moved, or restored SQL collection. AS/400 logical files may not be placed in an SQL collection because they cannot be described in the data dictionary.

You may create and own many collections.

Data Dictionary

A **data dictionary** is a set of tables containing object definitions.

SQL automatically creates a data dictionary when a collection is created. The dictionary is then automatically maintained by the system. You can work with data dictionaries by using the interactive data definition utility (IDDU), which is part of the OS/400 program. For more information on IDDU, see the *IDDU User's Guide*.

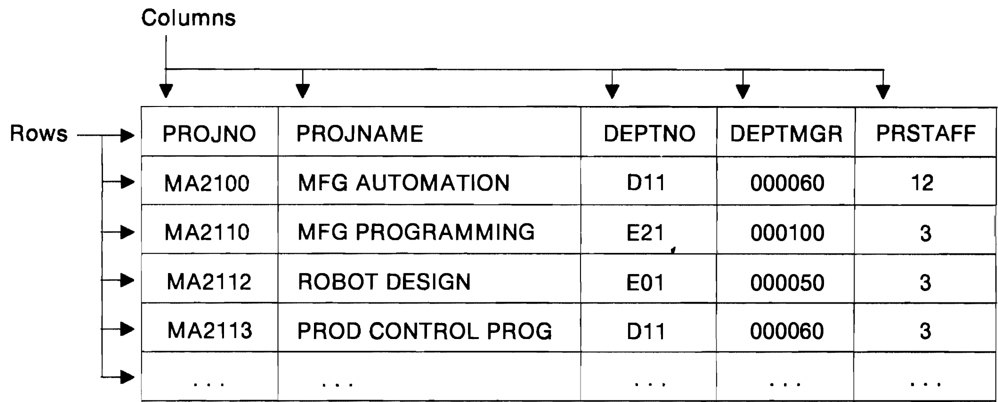
Journals and Journal Receivers

A **journal** and **journal receiver** are used to record changes to tables and views in the database. The journal and journal receiver are then used in processing SQL COMMIT and ROLLBACK statements. For more information on journaling, see the *Backup and Recovery Guide*.

Tables, Rows, and Columns

A **table** is a two-dimensional arrangement of data consisting of **rows** and **columns**. The row is the horizontal part containing one or more columns. The column is the vertical part containing one or more rows of data of one data type. All data for a column must be of the same type. A table in SQL is a nonkeyed physical file. See the section "Data Types" in the *SQL/400 Reference* manual for a description of data types.

The following is a sample SQL table:



PROJNO	PROJNAME	DEPTNO	DEPTMGR	PRSTAFF
MA2100	MFG AUTOMATION	D11	000060	12
MA2110	MFG PROGRAMMING	E21	000100	3
MA2112	ROBOT DESIGN	E01	000050	3
MA2113	PROD CONTROL PROG	D11	000060	3
...	...	...	...	...

SQL Data Types

When you create a table in SQL, you define each of its columns to hold one of the following types of data:

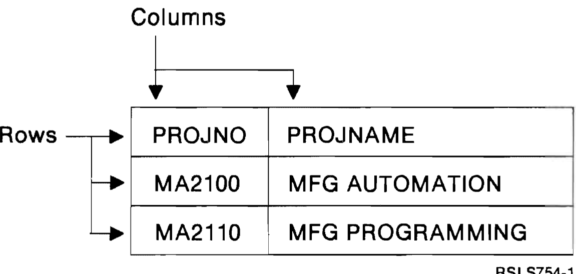
CHARACTER	Any EBCDIC character
DECIMAL	A packed decimal number with an implied decimal point
REAL	A short (4-byte) floating-point number in IEEE format
FLOAT	An 8-byte floating point number in IEEE format
INTEGER	A binary number with a precision of 31 bits
NUMERIC	A zoned decimal number with an implied decimal point
SMALLINT	A binary number with a precision of 15 bits

For more information on data types, see the section on "Data Types" in the *SQL/400 Reference* manual.

Views

A **view** appears like a table to an application program; however, a view contains no data. It is created over one or more tables. A view can contain all the columns of given tables or some subset of them, or can contain all the rows of given tables or some subset of them. The columns may be arranged differently in a view than they are in the tables from which they are taken. A view in SQL is a special form of a nonkeyed logical file.

The following figure shows a view created from the preceding example of an SQL table. Notice that the view is created only over the *PROJNO* and *PROJNAME* columns of the table and for rows MA2110 and MA2111.



Indexes

An SQL **index** is a collection of the data in the columns of a table that are logically arranged in either ascending or descending order. Each index contains a separate arrangement. An SQL/400 index is a keyed logical file.

The index is used by the system for faster data retrieval. Whether you create an index is optional. You can create any number of indexes. You can create or drop an index at any time. The index is automatically maintained by the system. However, because the indexes are maintained by the system, a large number of indexes can adversely affect the performance of applications that change the table.

Catalogs

An SQL **catalog** consists of a set of views and logical files based on:

- Two database files in QSYS (maintained by the AS/400 database manager) containing cross-reference information on:
  - The relationships between files and dictionaries
  - The relationships between files
- The set of data dictionary files in the collection containing object definitions

Catalog views only contain information about objects in one collection. The information in a catalog is about your SQL collection and its contents.

The catalog describes every table, view, index, and file in the collection and includes column definitions. However, there are four logical files existing in every collection that are not described in the catalog. The information can be queried like tables.

A catalog is automatically created when you create a collection. You cannot drop or explicitly change the catalog.

The views contained in an SQL catalog are named:

SYSCOLUMNS
SYSINDEXES
SYSKEYS
SYSTABLES
SYSVIEWDEP
SYSVIEWS

You can access information in the SQL catalog views by using normal SQL statements. The contents of each of the catalog views is described in the following section.

Catalog Views

The views contained in an SQL catalog are described in this section.

SYSCOLUMNS

The SYSCOLUMNS view contains one row for every column of each table and view in the SQL collection (including the columns of the SQL catalog). The following table describes the columns in the SYSCOLUMNS view:

Column Name	Data Type	Description
NAME	CHAR(10)	Name of the column
TBNAME	CHAR(10)	Name of the table or view that contains the column
TBCREATOR	CHAR(10)	The owner of the table or view
COLNO	SMALLINT	Numeric place of the column in the table or view, ordered from left to right
COLTYPE	CHAR(8)	Type of column: INTEGER — Large number SMALLINT — Small number FLOAT — Floating point; FLOAT, REAL, or DOUBLE PRECISION specified on the CREATE TABLE statement CHAR — Fixed-length character string DECIMAL — Packed decimal NUMERIC — Zoned decimal
LENGTH	SMALLINT	The length attribute of the column; or, in the case of a decimal, numeric, or nonzero precision binary column, its precision: 4 bytes — INTEGER 2 bytes — SMALLINT 8 bytes — FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION 4 bytes — FLOAT(n) where n = 1 to 24, or REAL Length of string — CHAR Precision of number — DECIMAL Precision of number — NUMERIC
SCALE	SMALLINT	Scale of numeric data (zero if not decimal, numeric, or nonzero precision binary)
NULLS	CHAR(1)	If the column can contain null values: N No Y Yes **Note:** This column always contains N.
UPDATES	CHAR(1)	If the column can be changed: N No Y Yes **Note:** The value is N anytime a column cannot be changed.
REMARKS	CHAR(254)	A character string you supply with the COMMENT ON statement

Column Name	Data Type	Description
DEFAULT	CHAR(1)	If the column has a default value (NOT NULL WITH DEFAULT): N No Y Yes
LABEL	CHAR(30)	A character string you supply with the LABEL ON statement
STORAGE	SMALLINT	The storage requirements for the column: 4 bytes — INTEGER 2 bytes — SMALLINT 8 bytes — FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION 4 bytes — FLOAT(n) where n = 1 to 24, or REAL Length of string — CHAR (Precision/2) + 1 — DECIMAL Precision of number — NUMERIC **Note:** This column supplies the storage requirements for all data types.
PRECISION	SMALLINT	The precision of numeric columns (zero if not numeric) **Note:** This column supplies the precision of all numeric data types, including single- and double-precision floating point.

SYSINDEXES

The SYSINDEXES view contains one row for every index in the SQL collection, including indexes on the SQL catalog. The following table describes the columns in the SYSINDEXES view:

Column Name	Data Type	Description
NAME	CHAR(10)	Name of the index
CREATOR	CHAR(10)	The owner of the index
TBNAME	CHAR(10)	Name of the table on which the index is defined
TBCREATOR	CHAR(10)	The owner of the table
TBDBNAME	CHAR(10)	Name of the SQL collection that contains the table on which the index is defined
UNIQUERULE	CHAR(1)	If the index is unique: D No (duplicates are allowed) U Yes
COLCOUNT	INTEGER	The number of columns in the key
DBNAME	CHAR(10)	Name of the SQL collection that contains the index

SYSKEYS

The SYSKEYS view contains one row for every column of an index type in the SQL collection (including the keys for the indexes on the SQL catalog). The following table describes the columns in the SYSKEYS view:

Column Name	Data Type	Description
IXNAME	CHAR(10)	Name of the index
IXCREATOR	CHAR(10)	The owner of the index
COLNAME	CHAR(10)	Name of the column of the key
COLNO	INTEGER	Numeric position of the column in the row
COLSEQ	INTEGER	Numeric position of the column in the key
ORDERING	CHAR(1)	Order of the column in the key: A Ascending D Descending

SYSTABLES

The SYSTABLES view contains one row for every table or view in the SQL collection (including the columns of the SQL catalog). The following table describes the columns in the SYSTABLES view:

Column Name	Data Type	Description
NAME	CHAR(10)	Name of the table or view
CREATOR	CHAR(10)	The owner of the table or view
TYPE	CHAR(1)	If the row describes a table or view: L Logical file P Physical file T Table V View
COLCOUNT	SMALLINT	Number of columns in the table or view
RECLENGTH	SMALLINT	The length of any record in the table
LABEL	CHAR(30)	A character string you supply with the LABEL ON statement
REMARKS	CHAR(254)	A character string you supply with the COMMENT ON statement
DBNAME	CHAR(10)	Name of the SQL collection that contains the table or view

SYSVIEWDEP

The SYSVIEWDEP view records the dependencies of views on tables (including the views of the SQL catalog). The following table describes the columns in the SYSVIEWDEP view:

Column Name	Data Type	Description
DNAME	CHAR(10)	Name of the view
DCREATOR	CHAR(10)	The owner of the view
BNAME	CHAR(10)	Name of a table the view is dependent on
BCREATOR	CHAR(10)	The owner of the table the view is based on
BDBNAME	CHAR(10)	Name of the SQL collection that contains the table that the view is dependent on
BTYPE	CHAR(1)	Type of object the view was based on: T Table V View

SYSVIEWS

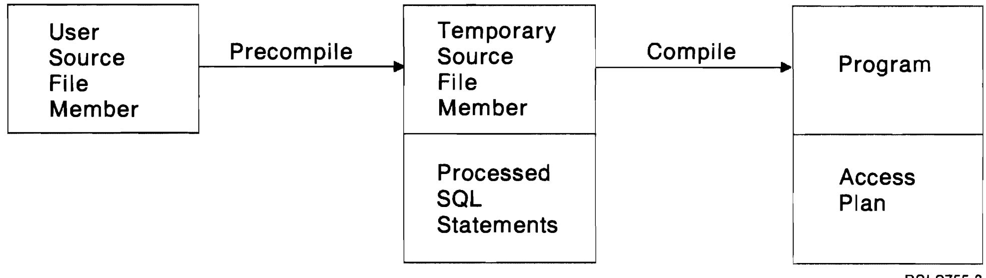
The SYSVIEWS view contains one or more rows for each view in the SQL collection (including the views of the SQL catalog). Each row contains a 70-character portion of the SQL CREATE VIEW statement that created the view. The following table describes the columns in the SYSVIEWS view:

Column Name	Data Type	Description
NAME	CHAR(10)	Name of the view
CREATOR	CHAR(10)	The owner of the view
SEQNO	INTEGER	Sequence number of this row; the first portion of the view is on row one and successive rows have increasing values of SEQNO
CHECK	CHAR(1)	Used only for compatibility with other SQL implementations. For the SQL/400 program, the value is always N.
TEXT	CHAR(70)	The text portion of the text of the CREATE VIEW statement

Compiled Application Program Objects

With the SQL/400 program, you do not have to manage any objects related to the application other than the original source and the resulting program.

The following shows the objects and steps that happen during the precompile and compile processes:



User Source File Member

A source file member contains the programmer's application language and SQL statements. Normally, you create and maintain the source file member by using the source entry utility (SEU), a part of the AS/400 Application Development Tools licensed program.

Temporary Source File Member

The temporary source file member is created in the library (QTEMP) by the precompile process (CRTSQLxxx commands)[2] and is automatically deleted by the system at job completion. A temporary source file member with the same name as the program name is added to QSQLTEMP by the precompile process. This member contains calls to:

- SQL run-time support, which has replaced embedded SQL statements
- Parsed and syntax checked SQL statements

By default, the host language compiler is called by the precompiler. For more information on precompilers, see Chapter 10.

Program

A program is the object created as a result of the compile process, which you can run.

An access plan is a set of internal structures and information that tells SQL how to run an embedded SQL statement in the most effective way. It is created only when a successful compile has occurred. Access plans are not created on the compile for SQL statements that reference a table or view that cannot be found or to which you are not authorized. The access plans for such statements will be created when the program is run. If, at that time, the table or view still cannot be found or you are still not authorized, a negative SQLCODE is returned. Access plans are stored and maintained in the program object.

2 The xxx in this command refers to the host language indicators CBL for COBOL/400, PLI for AS/400 PL/I, RPG for RPG III (part of RPG/400), and C for C/400.

Chapter 2. Working with SQL Collections, Tables, Views, Indexes, and Catalogs

This chapter describes how to create and work with SQL collections, tables, views, and indexes, and how to use catalogs in database design. It also describes using the COMMENT ON statement to describe the purpose of a table or view and any other special information about it.

The syntax and parameters for each of the SQL statements used in this chapter are described in detail in the *SQL/400 Reference* manual. The tables referred to in the examples are described in Appendix A.

A detailed description of how to use SQL statements and clauses in more complex situations is provided in Chapter 3.

Creating an SQL Collection

In the SQL/400 program, an SQL collection is the basic object in which tables, views, and indexes are placed. You must have authority to the CRTLIB and CRTDTADCT CL commands to run the CREATE COLLECTION statement. For more information on security, see *Security Concepts and Planning*.

A sample collection, named USER1, can be created with the following SQL statement:

```
CREATE COLLECTION USER1
```

Note: Running this statement causes several objects to be created and may take a few minutes.

After you have successfully created a collection, you can put tables, views, and indexes in it. For the purpose of supplying sample objects for the examples used in this manual, imagine that we have created the USER1 collection. The following sections describe adding tables and views to the USER1 collection.

Creating and Using a Table

The SQL CREATE TABLE statement is used to create a table and define the physical attributes of the columns in the table.

When this statement is processed, a new, empty table is created, containing the column information as defined. In the CREATE TABLE statement, the first parameter specifies the column name, the second parameter is the data type for that column, and the third parameter specifies if the column can be null. In the SQL/400 program, only NOT NULL or NOT NULL WITH DEFAULT is allowed.

Note: A table must be created in an SQL collection; it cannot be created in a library.

Other than the SQL return code, no other data is ever returned for the CREATE TABLE statement. The first-level message text of SQL return codes is described in Appendix B.

© Copyright IBM Corp. 1988, 1989

Creating the Department Table (TDEPT)

The sample department table describes each department in the company and specifies the department manager and the next higher department of authority.

You can create this table with the following interactive SQL statement:

```
CREATE TABLE USER1.TDEPT
    (DEPTNO     CHAR(3)   NOT NULL WITH DEFAULT,
     DEPTNAME   CHAR(36)  NOT NULL WITH DEFAULT,
     MGRNO      CHAR(6)   NOT NULL WITH DEFAULT,
     ADMRDEPT   CHAR(3)   NOT NULL WITH DEFAULT )
```

Note: You must type the complete statement before pressing Enter or you will get an error message.

Using the LABEL ON Statement for Defining Table Names and Column Headings

Sometimes the table name or column name does not clearly define the data when shown on an interactive display of the table. By using the LABEL ON statement, you can create a more descriptive label for the columns in addition to, or instead of, the table or column name.

In the interactive environment, the LABEL ON statements looks like this:

```
LABEL ON
  TABLE USER1.TDEPT IS 'Department Structure Table'

LABEL ON
  COLUMN USER1.TDEPT.ADMRDEPT IS 'Reports to Dept.'
```

where the table named TDEPT, when shown on an interactive display, will appear as *Department Structure Table* and the column named *ADMRDEPT* will have the heading *Reports to Dept*. The label for tables can be no more than 30 characters and the label for columns can be no more than 20 characters (blanks included). For more information about the LABEL ON statement, see the *SQL/400 Reference* manual.

Inserting Information into a Table

After you create a table, you can insert information (data values) into it by using the INSERT SQL statement (see "The INSERT Statement" on page 3-1 for more information on using this statement). In the interactive environment, the INSERT statements look like this:

```
INSERT INTO USER1.TDEPT
       (DEPTNO,
        DEPTNAME,
        MGRNO,
        ADMRDEPT )
  VALUES
       ('A00',
        'COMPUTER SERVICE DIV.',
        '000010',
        '   ' )
```

When the INSERT statement is run, a row is placed in the table. Note that a one-to-one correspondence exists between the column names specified in the INSERT clause and the data values specified in the VALUES clause. You must issue an INSERT statement for each row of the table. The sample collection now contains a table filled with data by issuing an INSERT statement for each row.

If COMMIT(*NONE) has been specified, no further action is required. Each insert is done at the time of the operation. However, if you specify COMMIT(*CHG) or COMMIT(*ALL), the insert is not guaranteed until the COMMIT statement is run. Conversely, the insert is backed out if you run a ROLLBACK statement.

Sample Department Table (TDEPT)

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
A00	COMPUTER SERVICE DIV.	000010	(blanks[1])
B01	PLANNING	000020	A00
C01	INFORMATION CENTER	000030	A00
D01	DEVELOPMENT CENTER	(blanks[1])	A00
E01	SUPPORT SERVICES	000050	A00
D11	MANUFACTURING SYSTEMS	000060	D01
D21	ADMINISTRATION SYSTEMS	000070	D01
E11	OPERATIONS	000090	E01
E21	SOFTWARE SUPPORT	000100	E01

[1] These entries were created with blank characters.

Getting Information from a Single Table

You can retrieve stored data by using the select-statement. The select-statement is the most complex of all SQL statements. This statement is composed of three main clauses:

1. The SELECT clause, which specifies those columns containing the desired data
2. The FROM clause, which specifies the table containing the columns with the desired data
3. The WHERE clause, which supplies a filter that determines which rows of data are retrieved

In addition to the three main clauses, there are several other clauses described in "Using Basic SQL Statements and Clauses" on page 3-1 and in the *SQL/400 Reference* manual, which affect the final form of returned data (for example, the ORDER BY clause).

In the interactive environment, the SQL statement looks like this:

```
SELECT EMPNO, LASTNAME, JOBCODE
   FROM USER1.TEMPL
   WHERE JOBCODE > 59
     AND SEX = 'F'
```

Note: The value being compared (in this case F) must be entered exactly as it appears in the table. For example, an f would not select any rows where the SEX column contained an F. However, in this example, adding the following OR clause would work.

```
AND SEX = 'F' OR SEX = 'f'
```

Refer to Appendix A to actually set up this environment and run the examples.

If the select-statement is successful, the result is one or more rows of the specified table.

The running of the above sample select-statement returns the following rows from the TEMPL table:

EMPNO	LASTNAME	JOBCODE
000010	HAAS	66
000030	KWAN	60

Note that only those rows are returned whose data values compare with the conditions specified by the predicates in the WHERE clause. Furthermore, the only data values returned are from the columns you explicitly specified in the SELECT clause. Data values of columns other than those explicitly identified are not returned.

Getting Information from More Than One Table

In addition to getting data from one table, SQL allows you to get information from columns contained in different tables. This operation is called a **join** operation. (For a more detailed description of the join operation, see "Joining Data from More Than One Table" on page 3-29). In SQL, a join operation is specified by placing the names of those tables you want to join together into the same FROM clause of a select-statement.

For example, consider the following select-statement, shown as you would enter it in the interactive environment in the SQL naming convention:

```
SELECT EMPNO, LASTNAME, USER1.TDEPT.DEPTNO, DEPTNAME
   FROM USER1.TEMPL, USER1.TDEPT
   WHERE JOBCODE > 59
     AND SEX = 'F'
     AND USER1.TEMPL.DEPTNO = USER1.TDEPT.DEPTNO
```

Note: Whenever the same column name exists in two or more tables being joined, the table name must be entered before the column name, connected with a period. For example, in the preceding select-statement, the column names DEPTNO and EMPNO are found in both tables.

With the system naming convention, you must use correlation names to achieve the same results. For example, in the case above, you would enter the following SQL statement:

```
SELECT EMPNO, LASTNAME, Y.DEPTNO, DEPTNAME
   FROM USER1/TEMPL X, USER1/TDEPT Y
   WHERE JOBCODE > 59
     AND SEX = 'F'
     AND X.DEPTNO = Y.DEPTNO
```

where X and Y are correlation names for TEMPL and TDEPT, respectively. For more information on correlation names, see the *SQL/400 Reference* manual.

If the select-statement is successful, the result is one or more rows of the specified table. The data values in the columns of the rows returned represent a composite of the data values contained in the two tables specified TEMPL and TDEPT.

The running of the above example select-statement returns the following columns:

EMPNO	LASTNAME	DEPTNO	DEPTNAME
000010	HAAS	A00	COMPUTER SERVICE DIV.
000030	KWAN	C01	INFORMATION CENTER

Changing Information in a Table

Using the UPDATE statement, you can change the data values in some of or all of the columns of a table. Also, you can limit the number of rows being changed during a single run by using the WHERE clause with the UPDATE statement. If the WHERE clause is absent, all of the rows in the specified table are changed. However, if the WHERE clause is highly restrictive, only one or a few rows are changed. (For more information on using the UPDATE statement and the WHERE clause, see "The UPDATE Statement" on page 3-3 and "The WHERE Clause" on page 3-8.)

For example, consider the following UPDATE statement, shown in the interactive environment, in which Michael L. Thompson's (employee no. 000020) telephone number is changed to 5678:

```
UPDATE USER1.TEMPL
   SET PHONENO = '5678'
   WHERE EMPNO = '000020'
```

The TEMPL table, shown in part, would be changed as follows:

EMPNO	FIRSTNME	MIDINIT	LASTNAME	DEPTNO	PHONENO
000010	CHRISTINE	I	HAAS	A00	3978
000020	MICHAEL	L	THOMPSON	B01	5678
000030	SALLY	A	KWAN	C01	4738

If you specify COMMIT(*NONE), no further action is required. Each change is then done at the time of the operation. However, if you specify COMMIT(*CHG) or COMMIT(*ALL), the change is not guaranteed until you run the COMMIT statement. Otherwise, the change is backed out if you run a ROLLBACK statement.

Deleting Information in a Table

The DELETE statement allows you to delete entire rows from a table because they no longer contain needed information. The DELETE statement allows you to use the WHERE clause to identify rows to be deleted during a single run. (For more information, see "The DELETE Statement" on page 3-4.)

For example, if Eileen W. Henderson left the company, the row containing information about this person must now be deleted. The following DELETE statement, shown in the interactive environment, looks like this:

```
DELETE
   FROM USER1.TEMPL
   WHERE EMPNO = '000090'
```

If you specify COMMIT(*NONE), no further action is required. Each delete is then done at the time of the operation. However, if you specify COMMIT(*CHG) or COMMIT(*ALL), the delete is not guaranteed until you run the COMMIT statement. Otherwise, the delete is backed out if you run a ROLLBACK statement.

Creating and Using a View

At times, you may find that no single table contains all the information you need, and that the data is scattered among several tables. Also, you may want to get only part of the data contained in a table and not a whole row or column of data, therefore limiting the access to certain kinds of data such as salary information. For these conditions, SQL lets you create a view.

You create a view in much the same way as you create a table. The view is defined by the CREATE VIEW statement. In order to create views, you must have SELECT authority on the tables on which views are based.

If you do not specify column names for the view, the column names will be the same as those for the table on which the view is based.

Note: A view must be created over tables or files that are in an SQL collection; a view cannot be created over tables or files in a library.

Creating a View on a Single Table

The following example shows how to create a view on a single table. The view is built on a table named USER1.TDEPT, which contains department data. The table has four columns, but the view is only on three of the columns: *DEPTNO*, *DEPTNAME*, and *MGRNO*. The order of the columns in the SELECT clause is the order in which they appear in the view. The CREATE VIEW statement looks like this:

```
CREATE VIEW USER1.VDEPT3 AS
  SELECT DEPTNO,DEPTNAME,MGRNO
  FROM USER1.TDEPT
```

In the example above, the columns of the view have the same name as the columns in the table, because no column list follows the view name. The following table is the result of running the SQL statement:

```
SELECT * FROM USER1.VDEPT3
```

DEPTNO	DEPTNAME	MGRNO
A00	COMPUTER SERVICE DIV.	000010
B01	PLANNING	000020
C01	INFORMATION CENTER	000030
D01	DEVELOPMENT CENTER	(blanks[2])
E01	SUPPORT SERVICES	000050
D11	MANUFACTURING SYSTEMS	000060
D21	ADMINISTRATION SYSTEMS	000070
E11	OPERATIONS	000090
E21	SOFTWARE SUPPORT	000100

[2] This entry was created with blank characters.

Creating a View Combining Data from More Than One Table

You can create a view that combines data from two or more tables by naming more than one table in the FROM clause. The combining operation is called a **join**. In the following example, the USER1.TEMPL table contains a column of manager's names called *LASTNAME*, and a column of employee serial numbers called *EMPNO*. These are joined with the *MGRNO* column of the USER1.TDEPT table. The CREATE VIEW statement looks like this:

```
CREATE VIEW USER1.VDEPTM AS
  SELECT USER1.TDEPT.DEPTNO,MGRNO,LASTNAME,ADMRDEPT
  FROM USER1.TDEPT,USER1.TEMPL
  WHERE USER1.TEMPL.EMPNO = USER1.TDEPT.MGRNO
```

When using the CREATE VIEW statement, the collection name specified could be any collection name that has already been created. For example, if you want the view above to reside in a collection named SMITH, enter the first line of the statement as:

```
CREATE VIEW USER1.VDEPTM AS
```

The following table is the result of running the SQL statement:

```
SELECT * FROM USER1.VDEPTM
```

DEPTNO	MGRNO	LASTNAME	ADMRDEPT[3]
A00	000010	HAAS	(blanks[4])
B01	000020	THOMPSON	A00
C01	000030	KWAN	A00
E01	000050	GEYER	A00
D11	000060	STEARN	D01
D21	000070	PULASKI	D01
E11	000090	HENDERSON	E01
E21	000100	SPENSER	E01

[3] If you typed the LABEL ON statement on page 2-2, this heading will be overridden by the heading specified on the LABEL ON statement and will read **Reports to Dept**.

[4] This entry was created with blank characters.

If you want to create the same view, including only those departments that report administratively to department A00, and you also want a different set of column names, your CREATE VIEW statement looks like this:

```
CREATE VIEW USER1.VDEPTMA00
  (DEPT,MGR,NAME,REPORTTO)
  AS
  SELECT USER1.TDEPT.DEPTNO,MGRNO,LASTNAME,ADMRDEPT
  FROM USER1.TEMPL,USER1.TDEPT
  WHERE USER1.TEMPL.EMPNO = USER1.TDEPT.MGRNO
  AND ADMRDEPT = 'A00'
```

The following table is the result of running the SQL statement:

```
SELECT * FROM USER1.VDEPTMA00
```

DEPT	MGR	NAME	REPORTTO[5]
B01	000020	THOMPSON	A00
C01	000030	KWAN	A00
E01	000050	GEYER	A00

[5] If you typed the LABEL ON statement on page 2-2, this heading will be overridden by the heading specified on the LABEL ON statement and will read **Reports to Dept**.

Notes on Using a View

When creating a view, the following restrictions must be considered:

- You cannot change, insert, or delete data in a view if it includes any of the following:
  - A FROM clause that uses more than one table (a join)
  - The FROM clause that identifies a read-only view
  - Any of the SQL column functions (SUM, MAX, MIN, AVG, or COUNT)
  - Elimination of duplicate rows (DISTINCT)
  - Grouping (GROUP BY), or HAVING

  In the above cases, you can get data from the views by means of the SQL SELECT statement, but you cannot use statements such as INSERT, UPDATE, or DELETE.

- You cannot insert a row in a view if:
  - The table on which the view is based has a column for which there is no default value, and that column is not in the view.
  - The view has a column resulting from an expression, a constant, or a function, and the column has been specified in the INSERT column list.

- You cannot update a column of a view that results from an expression, a constant, or a function.

- You cannot use the special register USER.

- You cannot use the scalar function LENGTH.

- You cannot use UNION or UNION ALL.

You can make changes to a table through a view even though the view does not contain the same number of columns or the same number of rows as the table on which it is based, provided that the columns not referred to were defined as NOT NULL WITH DEFAULT.

When you define a view on a table, it is like creating a new table, containing just the columns and rows you want.

You process the view as though it were a table, even though the view is totally dependent on one or more tables for data. The view has no data of its own and therefore requires no storage for the data. Because a view is derived from a table that exists in storage, when you update the view data, you are really updating data in the table. Therefore, views are automatically kept up-to-date as the tables they depend on are updated.

Views let you deal only with the data you need. A view reduces complexity and, at the same time, restricts access. When your application uses a view (instead of the table on which the view is based), it cannot access other rows or columns of the table that are not included in the view.

Working with Indexes

An index is used by the system for faster data retrieval. The following example creates an index over the column *LASTNAME* in the USER1.TEMPL table:

```
CREATE INDEX INX1 ON USER1.TEMPL (LASTNAME)
```

Any number of indexes can be created; however, because the indexes are maintained by the system, a large number of indexes can adversely affect performance. See Chapter 14 for more information on using indexes.

Note: An index can be created only over tables or files that are in an SQL collection.

Using the Catalog in Database Design

A catalog is automatically created when you create a collection. As the following examples show, you can display catalog information, but you cannot INSERT, DELETE, or UPDATE catalog information.

You must have SELECT privileges on the catalog views to run the following examples.

Getting Catalog Information about a Table

SYSTABLES contains a row for every table and view in the SQL collection. It tells you if the object is a table or view, the object name, the owner of the object, what SQL collection it is in, and so forth.

The following sample statement displays information for the USER1.TDEPT table:

```
SELECT *
  FROM USER1.SYSTABLES
  WHERE NAME = 'TDEPT'
```

Getting Catalog Information about a Column

SYSCOLUMNS contains a row for each column of every table and view in the collection.

The following sample statement displays all the column names in the USER1.TDEPT table:

```
SELECT *
  FROM USER1.SYSCOLUMNS
  WHERE TBNAME = 'TDEPT'
```

The result of the previous sample statements is a row of information for each column in the table. Some of the information is not visible because the width of the information is wider than the display screen.

For more information about each column, specify a select-statement like this:

```
SELECT NAME,TBNAME,COLTYPE,LENGTH,DEFAULT
  FROM USER1.SYSCOLUMNS
  WHERE TBNAME = 'TDEPT'
```

In addition to the column name for each column, the select-statement shows:

- The name of the table that contains the column
- The data type of the column
- The length attribute of the column
- If the column allows default values

The result looks like this:

NAME	TBNAME	COLTYPE	LENGTH	DEFAULT
DEPTNO	TDEPT	CHAR	3	Y
DEPTNAME	TDEPT	CHAR	36	Y
MGRNO	TDEPT	CHAR	6	Y
ADMRDEPT	TDEPT	CHAR	3	Y

Getting Catalog Information about Indexes

SYSINDEXES contains a row for every index in the collection. The following example gets information about the index INX1:

```
SELECT *
  FROM USER1.SYSINDEXES
  WHERE NAME = 'INX1'
```

The previous example displays a single row of information about a particular index. However, a table can have more than one index. To display information about all the indexes on a table, write a statement like this:

```
SELECT *
  FROM USER1.SYSINDEXES
  WHERE TBNAME = 'TEMPL'
```

Note: In this example, there happens to be only one index over TEMPL.

Getting Catalog Information about Views

For every view you create in the collection, information is stored in the catalog views. This is what happens in the catalog, following a CREATE VIEW:

- A row is inserted into SYSTABLES.
- A row is inserted into SYSCOLUMNS for each column of the view.
- One or more rows are inserted into SYSVIEWS to record the text of the CREATE VIEW statement.
- For each table on which the view is dependent, a row is inserted into SYSVIEWDEP to record what the view is dependent on.

Building a View over Catalog Views

You can build a view on one or more of the catalog views containing information about your own tables and views. The following example creates a view of a user's tables from the SYSTABLES view:

```
CREATE VIEW MYTABLES AS
  SELECT * FROM USER1.SYSTABLES
  WHERE CREATOR = 'USER1'
```

Using COMMENT ON

After you create a table or view, you can supply information about it for future reference, such as the purpose of the table or view, who uses it, and anything unusual or special about it. You can also include similar information about each column of the table or view. Your comment must not be more than 254 bytes.

A comment is especially useful if your names do not clearly indicate the contents of the columns or tables. In that case, use a comment to describe the specific contents of the column or table.

If you include REMARKS in the list of column names you get from SYSCOLUMNS, any comments you had included for the columns are displayed. Both SYSTABLES and SYSCOLUMNS have a column for REMARKS.

An example of using COMMENT ON follows:

```
COMMENT ON TABLE USER1.TEMPL IS
  'Employee table.  Each row in this table represents
   one employee of the company.'
```

Getting Comments

After running a COMMENT ON statement, your comments are stored in the *REMARKS* column of SYSTABLES or SYSCOLUMNS. (If the indicated row had already contained a comment, the old comment is replaced by the new one.) The following example gets the comments added by the COMMENT ON statement in the previous example:

```
SELECT REMARKS
  FROM USER1.SYSTABLES
  WHERE NAME = 'TEMPL'
```

Chapter 3. SQL Coding Techniques

This chapter describes coding techniques for:

- Using basic SQL statements and clauses
- Using the UNION keyword to combine subselects
- Using a cursor
- Performing complex search conditions
- Joining data from more than one table
- Inserting multiple rows into a table
- Inserting default values into columns

You can embed SQL statements in an application program or issue them interactively, using interactive SQL (described in Chapter 11).

Using Basic SQL Statements and Clauses

This section shows the basic SQL statements and clauses that retrieve, update, delete, and insert data into tables and views. The SQL statements used are SELECT, UPDATE, DELETE, and INSERT. Also, FETCH can be used in an application program to access data. Examples using these SQL statements are supplied to help you develop SQL statements. Detailed syntax and parameter descriptions for SQL statements are given in the *SQL/400 Reference* manual.

You can write SQL statements on one line or on many lines. The rules for the continuation of lines are the same as those of the host language (the language the program is written in), except for C. In C, when using line continuation, an SQL statement can be split wherever a blank can occur, except within a string constant.

Notes:

1. The SQL statements described in this chapter can be run on SQL tables and views and database physical and logical files. The tables and views can be either in an SQL collection or in a library.
2. Character strings specified in an SQL statement (such as those used with WHERE or VALUES clauses) are case sensitive; that is, uppercase characters must be entered in uppercase and lowercase characters must be entered in lowercase.

The INSERT Statement

You can use the INSERT statement to add new rows to a table or view by doing one of the following:

- Specifying values in the INSERT statement for columns of the single row to be added.
- Including a subselect in the INSERT statement to tell SQL what data for the new row is contained in another table or view. "Inserting Multiple Rows into a Table" on page 3-32, explains how to use the subselect within an INSERT statement to add multiple rows to a table.

Note: Because views are built on tables and actually contain no data, working with views can be complicated. See "Notes on Using a View" on page 2-9 for more information on inserting information into a view.

© Copyright IBM Corp. 1988, 1989

For every row you insert, you must supply a value for each NOT NULL column of a table that does not have a default value. The INSERT statement for adding a row to a table or view may look like this:

```
EXEC SQL
  INSERT INTO table-name
    [(column-name-1 [, column-name-2] ... )]
    VALUES (value-1 [, value-2] ... )
END-EXEC.
```

The INTO clause names the columns for which you specify values. The VALUES clause specifies a value for each column named in the INTO clause.

For example, suppose you want to add a new employee to the USER1.TEMPL table:

```
EXEC SQL
  INSERT INTO USER1.TEMPL
    (EMPNO, FIRSTNME, MIDINIT, LASTNAME, DEPTNO)
    VALUES (:PGM-SERIAL, :PGM-NAME1,
      :PGM-NAME2, :PGM-NAME3, :PGM-DEPT)
END-EXEC.
```

First, you might supply values for only those columns that have no default values. SQL inserts the new values (contained in host variables) into the USER1.TEMPL table in the order in which you list them. The first value (from PGM-SERIAL) is put into the first specified column (*EMPNO*), the second value (from PGM-NAME1) is put into the second specified column (*FIRSTNME*), and so on. If you specify fewer column names than there are columns in the row, SQL places default values in the remaining columns.

You must provide a value in the VALUES clause for each column named in an INSERT statement's column list. You can name all columns for which you are providing values, or you can omit the column name list.

It is a good idea to name all columns into which you are inserting values because:

Your source statements are more descriptive.
You can verify that you are giving the values in order.
You have better data independence.

When your program attempts to insert a row that duplicates another row already in the table, an error might occur:

- If the table has a unique index, the row is not inserted. Instead, SQL returns an SQLCODE of −803.
- If the table does not have a unique index, the row can be inserted without error.

If SQL finds an error while running the INSERT statement, it stops inserting data. No rows in the table are inserted (rows already inserted, if any, are deleted), if COMMIT(*ALL) or COMMIT(*CHG) is specified. If you specify COMMIT(*NONE), any rows already inserted are *not* deleted.

If the row is inserted without error, the SQLERRD(3) field of the SQLCA will have a value of 1.

Note: INSERT with subselect may have more than one row to insert. The number of rows inserted is reflected in SQLERRD(3).

The UPDATE Statement

To change the data in a table, use the UPDATE statement. With the UPDATE statement, you can change the value of one or more columns in each row that satisfies the search condition of the WHERE clause. The result of the UPDATE statement is one or more changed column values in zero or more rows of a table (depending on how many rows satisfy the search condition specified in the WHERE clause). The UPDATE statement looks like this:

```
EXEC SQL
  UPDATE table-name
    SET column-name-1 = value-expression
    [, column-name-2 = value-expression ] ...
    WHERE search-condition ...
END-EXEC.
```

For example, suppose an employee has been relocated. To update several items of the employee's data in the USER1.TEMPL table to reflect the move, you can specify:

```
EXEC SQL
  UPDATE USER1.TEMPL
    SET JOBCODE = :PGM-CODE,
    PHONENO = :PGM-PHONE
    WHERE EMPNO = :PGM-SERIAL
END-EXEC.
```

First, name the table or view, then provide a `column-name = value-expression` pair for each column you want to update. Use the SET clause to specify the new value for a column. The SET clause names the columns you want updated and provides the values you want them changed to. The value-expression you specify can be:

A **column name.** Replace the column's current value with the contents of another column in the same row.

A **constant.** Replace the column's current value with the value provided in the SET clause.

A **host variable.** Replace the column's current value with the contents of the host variable.

A **special register.** Replace the column's current value with a special register value (USER).

An **expression.** Replace the column's current value with the value that results from an expression.

Next, identify the rows to be updated:

- To update a single row, use a WHERE clause that locates one, and only one, row.
- To update several rows, use a WHERE clause that locates only the rows you want to update.

You can omit the WHERE clause; if you do, SQL updates each row in the table or view with the values you supply.

If SQL finds an error while running your UPDATE statement, it stops updating and returns a negative SQLCODE. If you specify COMMIT(*ALL) or COMMIT(*CHG), no rows in the table are changed (rows already changed, if any, are restored to their

previous values). If COMMIT(*NONE) is specified, any rows already changed are *not* restored to previous values.

If SQL cannot find any rows that satisfy the search condition, an SQLCODE of +100 is returned.

Note: UPDATE with WHERE clause may have updated more than one row. The number of rows updated is reflected in SQLERRD(3).

The DELETE Statement

To remove rows from a table, use the DELETE statement. When you DELETE a row, you remove the entire row. DELETE does not remove specific columns from the row. The result of the DELETE statement is the removal of zero or more rows of a table (depending on how many rows satisfy the search condition specified in the WHERE clause). If you omit a WHERE clause from a DELETE statement, SQL removes all the rows of the table. The DELETE statement looks like this:

```
DELETE FROM table-name
  WHERE search-condition ...
```

For example, suppose department D11 was moved to another place. You want to delete each row in the USER1.TEMPL table with a DEPTNO value of D11 as follows:

```
EXEC SQL
  DELETE FROM USER1.TEMPL
    WHERE DEPTNO = 'D11'
END-EXEC.
```

When this statement is run, SQL deletes any row from the USER1.TEMPL table that meets the search condition.

The WHERE clause tells SQL which rows you want to delete from the table. SQL deletes all the rows that satisfy the search condition from the base table. You can omit the WHERE clause, but you will probably want to include one, because a DELETE statement without a WHERE clause deletes all the rows from the table or view. To delete a table definition as well as the table contents, issue the DROP statement (described in the *SQL/400 Reference* manual).

If SQL finds an error while running your DELETE statement, it stops deleting data and returns a negative SQLCODE. If you specify COMMIT(*ALL) or COMMIT(*CHG), no rows in the table are deleted (rows already deleted, if any, are restored to their previous values). If COMMIT(*NONE) is specified, any rows already deleted are *not* restored to their previous values.

If SQL cannot find any rows that satisfy the search condition, then an SQLCODE of +100 is returned.

Note: DELETE with WHERE clause may have deleted more than one row. The number of rows deleted is reflected in SQLERRD(3).

The SELECT INTO Statement

You can use the SELECT INTO statement[1] to retrieve a specific row (for example, the row for an employee). The format and syntax shown here are very basic. SELECT INTO statements can be more varied than the examples presented in this chapter. A SELECT INTO statement specifies six items of information:

1. The name of each column you want
2. The name of each host variable used to contain retrieved data
3. The name of the table or view that contains the data
4. A search condition to uniquely identify the row that contains the information you want
5. The name of each column used to group your data
6. A search condition that uniquely identifies a group that contains the information you want

A SELECT INTO statement looks like this:

```
EXEC SQL
  SELECT column names
    INTO host variables
    FROM table or view name
    WHERE search condition
    GROUP BY column names
    HAVING search condition
END-EXEC.
```

The INTO clause names the host variables (variables in your program used to contain retrieved column values). The value of the first column specified in the SELECT clause is put into the first host variable named in the INTO clause; the second value is put into the second host variable, and so on.

The INTO clause is part of the SELECT INTO statement because we are assuming the WHERE clause can be satisfied by only one row. For example, each row in the USER1.TEMPL table has a unique EMPNO (employee number). Therefore, the result of the SELECT INTO statement is assumed to be either one or zero rows. Finding more than one row is an error, although one row will be returned.

If you want several rows to be the result of a select-statement, use a DECLARE CURSOR statement to select the rows, followed by a FETCH statement to move the column values into host variables one row at a time. A technique for using cursors is described in "Using a Cursor" on page 3-20.

The FROM clause names the table (or view) that contains the data you are interested in.

[1] For the complete syntax of the SELECT statement, see the *SQL/400 Reference* manual.

For example, assume that each department listed in the USER1.TDEPT table has a unique department number. You want to retrieve the department name and manager number from the USER1.TDEPT table for department C01. To do this, your program can issue:

```
EXEC SQL
  SELECT DEPTNAME, MGRNO
    INTO :PGM-DEPTNAME, :PGM-MGRNO
    FROM USER1.TDEPT
    WHERE DEPTNO = :PGM-DEPT
END-EXEC.
```

In this example, PGM-DEPT has been set to C01. When the statement is run, the result is one row:

PGM-DEPTNAME	PGM-MGRNO
INFORMATION CENTER	000030

If SQL is unable to find a row that satisfies the search condition, an SQLCODE of +100 is returned.

If SQL finds an arithmetic expression error while running your statement, one of two things occurs:

- If the arithmetic expression error occurs in the SELECT list or during the evaluation of an argument of a column or scalar function in the SELECT list, and an indicator variable is provided to the expression in error:
  - SQL returns a −2 for the indicator variable corresponding to the expression in error.
  - SQL returns all valid data for that row.
  - SQL returns an SQLCODE of +802.
- If the arithmetic expression error occurs under any other circumstances, SQL stops retrieving rows and returns an SQLCODE of −802 in the SQLCA.

If SQL finds a numeric conversion error while running your select-statement, one of two things occurs:

- If the numeric conversion error occurs while converting a numeric value into the host variable and an indicator variable is provided with the host variable:
  - SQL returns a −2 for the host variable corresponding to the expression in error.
  - SQL returns all valid data for that row.
  - SQL returns an SQLCODE of +304.
- If the numeric conversion error occurs while converting a numeric value into the data type of the host variable and an indicator variable is not provided with the host variable, SQL stops running your statement and returns an SQLCODE of −304.

In the case of SQLCODEs of −304 or −802, the SQLCA reports only the first conversion error or arithmetic error detected. However, the indicator variable corresponding to each results column, having arithmetic and/or conversion errors, will be set to −2.

If SQL finds any other errors while running your select-statement, a negative SQLCODE is returned.

If SQL finds that the retrieved character column value length is too large to be saved in a host variable, SQL will:

- Truncate the data while assigning the value to the host variable.
- Set SQLWARN0 and SQLWARN1 in the SQLCA to the value W.
- Set SQLWARN0 to W and SQLWARN1 to N when using C/400, for a varying length null-terminated character variable, if only the null-terminator is truncated.
- Set the indicator variable, if provided, to the length of the value before truncation.

Specifying the Columns You Want

With the SELECT clause (the first part of a select-statement), you specify the name of each column you want to retrieve. For example:

```
SELECT EMPNO, LASTNAME, DEPTNO
  :
```

You can specify that only one column be retrieved, or as many as 8000 columns. The value of each column you name is retrieved in the order specified in the SELECT clause.

If you want to retrieve all columns (in the same order as they appear in the row), use an asterisk (*) instead of naming the columns:

```
SELECT *
  :
```

You can specify a SELECT * clause in a program, but this is not recommended because, if the table definition changes, your program might reference columns for which no receiving host variables are defined.

When using the select-statement in an application program, list the column names to give your program more data independence. There are two reasons for this:

1. When you look at the source code statement, you can easily see the one-to-one correspondence between the column names in the SELECT clause and the host variables named in the INTO clause.
2. If a column is added to a table or view you access and you use "SELECT * ...," the INTO clause does not have a matching host variable named for the new column. The extra column causes you to get a warning (not an error) in the SQLCA (SQLWARN4 will contain a "W"; in RPG this field is SQLWN4).

Processing Data in a View

You can retrieve data from a view in exactly the same way you retrieve data from a table. However, there are several restrictions when you attempt to update, insert, or delete data in a view. These restrictions are described in "Notes on Using a View" on page 2-9.

The WHERE Clause

The WHERE clause specifies a search condition that identifies the row or rows you want to retrieve, update, or delete. The number of rows you process with an SQL statement then depends on the number of rows that satisfy the WHERE clause **search condition**. A search condition consists of one or more **predicates**. A predicate specifies a test that you want SQL to apply to a specified row or rows of a table.

In the following example, DEPTNO = 'C01' is a predicate, DEPTNO and 'C01' are expressions, and the equal sign (=) is a comparison operator. Note that character values are enclosed in apostrophes ('); numeric values are not. This applies to all literal values wherever they are coded within an SQL statement. For example, to specify that you are interested in the rows where the department number is C01, you would say:

```
...
WHERE DEPTNO = 'C01'
```

In this case, the search condition consists of one predicate: DEPTNO = 'C01'.

Using Expressions in the WHERE Clause

An expression in a WHERE clause names or specifies something you want to compare to something else. Each expression, when resolved by SQL, has a character string or a numeric value. The expressions you specify can be:

- A **column name** names a column. For example:

  ```
  ...
  WHERE EMPNO = '000200'
  ```

  EMPNO names a column that is defined as a 6-byte character value. Equality comparisons (that is, X = Y or X ¬= Y) can be performed on character data. Other types of comparisons can also be evaluated for character data, based on the binary collating sequence.

  However, you cannot compare character strings to numbers. You also cannot perform arithmetic operations on character data (even though EMPNO is a character string that appears to be a number).

- An **expression** identifies two values that are added (+), subtracted (—), multiplied (*), divided (/), or concatenated (||) to result in a value. The operands of an expression can be:

  A constant (that is, a literal value)
  A column
  A host variable
  A value returned from a function
  A special register
  Another expression

  For example:

  ```
  ...
  WHERE JOBCODE + EDUCLVL > 70
  ```

JOBCODE names a column that is defined as a 3-digit packed decimal value (DECIMAL(3)). *EDUCLVL* names a column that is defined as a halfword integer value (SMALLINT).

- A **constant** specifies a literal value for the expression. For example:

```
...
WHERE 40000 < SALARY
```

  SALARY names a column that is defined as an 8-digit packed decimal value (DECIMAL(8,2)).

- A **host variable** identifies a variable in an application program. For example:

```
...
WHERE EMPNO = :EMP
```

- A **special register** identifies a special value generated by the database manager. For example:

```
...
WHERE LASTNAME = USER
```

A search condition need not be limited to two column names or constants separated by an arithmetic or comparison operator. You can develop a complex search condition that specifies several predicates separated by AND and OR, names and constants. No matter how complex the search condition, it will supply a TRUE or FALSE value when evaluated against a row. There is also an *unknown* truth value, which is effectively false. That is, if the value of a row is null, this null value is not returned as a result of a search because it is not less than, equal to, or greater than the value specified in the search condition. More complex search conditions and predicates are described in "Performing Complex Search Conditions" on page 3-26.

To fully understand the WHERE clause, you need to know how SQL evaluates search conditions and predicates, and compares the values of expressions. This topic is discussed in the *SQL/400 Reference* manual.

Comparison Operators: The SQL comparison operators are:

Operator	Meaning
=	Equal to
¬= or <>	Not equal to
<	Less than
>	Greater than
<= or ¬>	Less than or equal to (or not greater than)
> = or ¬<	Greater than or equal to (or not less than)

The NOT Keyword

You can precede a predicate with the NOT keyword to specify that you want the opposite of the predicate's value (that is, TRUE if the predicate results are FALSE, or vice versa). For example, to indicate that you are interested in all employees except those working in department C01, you could say:

```
...
WHERE NOT DEPTNO = 'C01'
```

which is equivalent to:

```
...
WHERE DEPTNO ¬= 'C01'
```

Note: You cannot make the NOT keyword part of a comparison operator: NOT must precede a predicate.

The USER Special Register

When an SQL statement containing the USER special register is run, SQL replaces USER with the authorization ID of the person running the program.

There are two restrictions on using the USER special register:

1. USER cannot be used in a CREATE VIEW statement.
2. The authorization ID of the person running the program cannot be longer than 8 characters.

Multiple Predicates

A WHERE clause can contain several predicates, each of which is separated by a **logical connective**, AND or OR. For example, to locate all male employees in department D21, you can specify:

```
...
WHERE SEX = 'M' AND DEPTNO = 'D21'
```

You can precede a predicate with the NOT keyword to specify the opposite of the predicate. However, NOT applies only to the predicate it precedes, not to all predicates in the WHERE clause. For example, to specify all women who work in department D11:

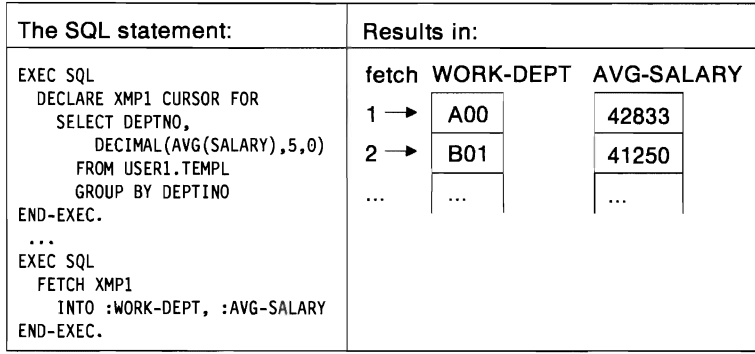
```
...
WHERE NOT SEX = 'M'
      AND DEPTNO = 'D11'
```

The GROUP BY Clause

Without a GROUP BY clause, the application of SQL column functions yields *one* row. When GROUP BY is used, the function is applied to *each* group, thereby yielding as many rows as there are groups.

The GROUP BY clause allows you to find the characteristics of groups of rows rather than individual rows. When you specify a GROUP BY clause, SQL divides the selected rows into groups such that the rows of each group have matching values in one or more columns. Next, SQL processes each group to produce a single-row result for the group. You can specify one or more columns in the GROUP BY clause to group the rows. The items you specify in the SELECT statement are properties of each group of rows, not properties of individual rows in a table or view.

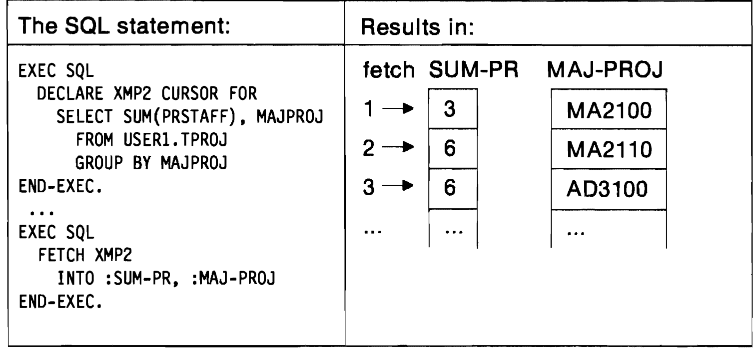
For example, the USER1.TEMPL table has several sets of rows, and each set consists of rows describing members of a specific department. To find the average salary of people in each department, you could issue:



The result is several rows, one for each department.

Note: Grouping the rows does not mean ordering them. Grouping puts each selected row in a group, which SQL then processes to derive characteristics of the group. Ordering the rows puts all the rows in the results table in ascending or descending collating sequence. ("The ORDER BY Clause" on page 3-14 describes how to do this.)

When you use GROUP BY, you name the columns you want SQL to use to group the rows. For example, suppose you want a list of the number of people working on each major project described in the USER1.TPROJ table. You could issue:



The SQL statement:

```
EXEC SQL
  DECLARE XMP2 CURSOR FOR
    SELECT SUM(PRSTAFF), MAJPROJ
      FROM USER1.TPROJ
      GROUP BY MAJPROJ
END-EXEC.
 ...
EXEC SQL
  FETCH XMP2
    INTO :SUM-PR, :MAJ-PROJ
END-EXEC.
```

Results in:

fetch	SUM-PR	MAJ-PROJ
1 →	3	MA2100
2 →	6	MA2110
3 →	6	AD3100
...	...	...

RSLS790-0

The result is a list of the company's current major projects and the number of people working on each project.

You can also specify that you want the rows grouped by more than one column. For example, you could issue a select-statement to find the average salary for men and women in each department, using the USER1.TEMPL table. To do this, you could issue:

The SQL statement:

```
EXEC SQL
  DECLARE XMP3 CURSOR FOR
    SELECT DEPTNO, SEX,
        DECIMAL(AVG(SALARY),5,0)
      FROM USER1.TEMPL
      GROUP BY DEPTNO, SEX
END-EXEC.
 ...
EXEC SQL
  FETCH XMP3
    INTO :DEPT, :SEX,
      :AVG-WAGES
END-EXEC.
```

Results in:

fetch	DEPT	SEX	AVG-WAGES
1 →	A00	F	52750
2 →	A00	M	37875
3 →	B01	M	41250
4 →	C01	F	30156
...	...	...	...

RSLS791-0

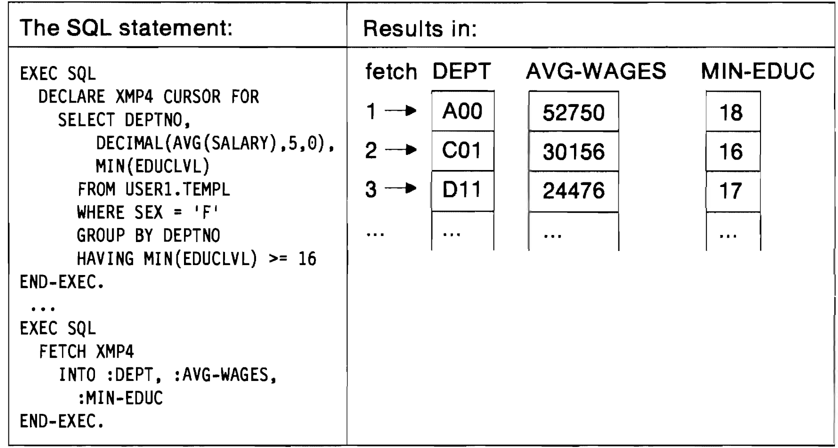
Because you did not include a WHERE clause in this example, SQL examines and process all rows in the USER1.TEMPL table. The rows are grouped first by department number and next (within each department) by sex before SQL derives the average SALARY value for each group.

The HAVING Clause

You can use the HAVING clause to specify a search condition for the groups selected based on a GROUP BY clause. The HAVING clause says that you want *only* those groups that satisfy the condition in that clause. Therefore, the search condition you specify in the HAVING clause must test properties of each group rather than properties of individual rows in the group.

The HAVING clause follows the GROUP BY clause and can contain the same kind of search condition you can specify in a WHERE clause. In addition, you can specify column functions in a HAVING clause. For example, suppose you wanted to retrieve the average salary of women in each department. To do this, you would use the AVG column function and group the resulting rows by DEPTNO and specify a WHERE clause of SEX = 'F'.

To specify the condition that you want this data only when all the employees in the selected department who have an education level equal to or greater than 16 (a college graduate), use the HAVING clause. The HAVING clause tests a property of the group. In this case, the test is on MIN(EDUCLVL), which is a group property:



The SQL statement:

```
EXEC SQL
  DECLARE XMP4 CURSOR FOR
    SELECT DEPTNO,
        DECIMAL(AVG(SALARY),5,0),
        MIN(EDUCLVL)
      FROM USER1.TEMPL
      WHERE SEX = 'F'
      GROUP BY DEPTNO
      HAVING MIN(EDUCLVL) >= 16
END-EXEC.
 ...
EXEC SQL
  FETCH XMP4
    INTO :DEPT, :AVG-WAGES,
      :MIN-EDUC
END-EXEC.
```

Results in:

fetch	DEPT	AVG-WAGES	MIN-EDUC
1 →	A00	52750	18
2 →	C01	30156	16
3 →	D11	24476	17
...	...	...	...

RSLS792-0

You can use multiple predicates in a HAVING clause by connecting them with AND and OR, and you can use NOT for any predicate of a search condition.

The ORDER BY Clause

You can specify that you want selected rows retrieved in a particular order, sorted by ascending or descending collating sequence of a column's value, with the ORDER BY clause. You can use an ORDER BY clause as you would a GROUP BY clause: specify the name of the column or columns you want SQL to use when retrieving the rows in a collated sequence.

Instead of naming the columns to order the results, you can use a number. For example, ORDER BY 3 specifies that you want the results ordered by the *third* column of the results table, as specified by the select-statement. Use a number to order the rows of the results table when the sequencing value is the result of an expression, column function, or something other than a column name in the select-statement.

For example, to retrieve the names and department numbers of female employees listed in the alphanumeric order of their department numbers, you could use this select-statement:

The SQL statement:

```
EXEC SQL
  DECLARE XMP5 CURSOR FOR
    SELECT LASTNAME, DEPTNO
      FROM USER1.TEMPL
      WHERE SEX = 'F'
      ORDER BY DEPTNO
END-EXEC.
 ...
EXEC SQL
  FETCH XMP5
    INTO :PGM-NAME3, :DEPT
END-EXEC.
```

Results in:

fetch	PGM-NAME3	DEPT
1→	HAAS	A00
2→	KWAN	C01
3→	QUINTANA	C01
4→	NICHOLLS	C01
5→	PIANKA	D11
6→	SCOUTTEN	D11
7→	LUTZ	D11
8→	PULASKI	D21
9→	JOHNSON	D21
10→	PEREZ	D21
11→	HENDERSON	E11
12→	SCHNEIDER	E11

RSLS793-0

Note: All columns named in the ORDER BY clause must also be named in the SELECT statement.

You can also specify whether you want SQL to collate the rows in ascending (ASC) or descending (DESC) sequence. An ascending collating sequence is the default. In the above select-statement, SQL first returns the row with the lowest department number (alphabetically and numerically), followed by rows with higher department numbers. To order the rows in descending collating sequence based on the department number, specify:

```
...
ORDER BY DEPTNO DESC.
```

As with GROUP BY, you can specify a secondary ordering sequence (or several levels of ordering sequences) as well as a primary one. In the example above, you might want the rows ordered first by department number, and within each department, ordered by employee name. To do this, specify:

```
...
ORDER BY DEPTNO, LASTNAME.
```

Note: If you intend to update a column or delete a row, you cannot include a GROUP BY or HAVING clause in the SELECT statement within a DECLARE CURSOR statement. (The DECLARE CURSOR statement is described in "Using a Cursor" on page 3-20.)

Using the UNION Keyword to Combine Subselects

Using the UNION keyword, you can combine two or more subselects to form a single select-statement. When SQL encounters the UNION keyword, it processes each sub-select to form an interim result table, then it combines the interim result table of each subselect and deletes duplicate rows to form a combined result table. You use UNION to merge lists of values from two or more tables. You can use any of the clauses and techniques you have learned so far when coding select-statements, including ORDER BY.

You can use UNION to eliminate duplicates when merging lists of values obtained from several tables. For example, you can obtain a combined list of employee numbers that includes:

- People in department D11
- People whose assignments include projects MA2112, MA2113, and AD3111

The combined list is derived from two tables and contains no duplicates. To do this, specify:

```
MOVE 'D11' TO WORK-DEPT.
 ...
  EXEC SQL
  DECLARE XMP6 CURSOR FOR
  SELECT EMPNO
    FROM USER1.TEMPL
    WHERE DEPTNO = :WORK-DEPT
  UNION
  SELECT EMPNO
    FROM USER1.TEMPRACT
    WHERE PROJNO = 'MA2112' OR
          PROJNO = 'MA2113' OR
          PROJNO = 'AD3111'
  ORDER BY 1
END-EXEC.
 ...
EXEC SQL
  FETCH XMP6
    INTO :EMP-NUMBER
END-EXEC.
```

To better understand what results from these SQL statements, imagine that SQL goes through the following process:

Step 1: SQL processes the first SELECT statement:	Which results in an interim result table:
... SELECT EMPNO FROM USER1.TEMPL WHERE DEPTNO = 'D11' ...	(from USER1.TEMPL) 000060 000150 000160 000170 ...

Step 2: SQL processes the second SELECT statement:	Which results in another interim result table:
... SELECT EMPNO FROM USER1.TEMPRACT WHERE PROJNO = 'MA2112' OR PROJNO = 'MA2113' OR PROJNO = 'AD3111' ...	(from USER1.TEMPRACT) 000170 000190 000180 ...

Step 3: SQL combines the two interim result tables:	Which results in a combined result table with values in ascending sequence:
EXEC SQL DECLARE XMP6 CURSOR FOR SELECT ... UNION SELECT ... ORDER BY 1 END-EXEC. ... EXEC SQL FETCH XMP6 INTO :EMP-NUMBER END-EXEC.	fetch EMP-NUMBER 1 → 000060 2 → 000150 3 → 000160 4 → 000170 5 → 000180 ... ...

RSLS794-0

When you use UNION:

- Any ORDER BY clause must appear after the last subselect that is part of the union. In this example, the results are sequenced on the basis of the first selected column, *EMPNO*. The ORDER BY clause specifies that the combined result table is to be in collated sequence.

  Note: To specify the columns that SQL should order the results by, use numbers (in a union, you cannot use column names for this). The number refers to the position of the expression in the list of expressions you include in your subselects.

- You cannot use UNION when creating a view.

To identify which subselect each row is from, you can include a constant at the end of the select list of each subselect in the union. When SQL returns your results, the last column contains the constant for the subselect that is the source of that row. For example, you can specify:

```
SELECT A, B, 'A1' ... UNION SELECT X, Y, 'B2'
```

When a row is presented to your program, it includes a value (either A1 or B2) to indicate the table that is the source of the row's values. If the column names in the union are different, SQL uses the set of column names specified in the first subselect when interactive SQL displays or prints the results, or in the SQLDA resulting from processing an SQL DESCRIBE statement.

In the example above, SQL uses A and B.

Restrictions for the Length and Data Type of Columns

When you use UNION, the lengths and data types of the columns named in the SELECT statements must be comparable; any necessary conversions are made to produce the values of the result table. For example, if you specify:

```
SELECT EMPNO ... UNION SELECT DEPTNO ...
```

where *EMPNO* is CHAR(6) and *DEPTNO* is CHAR(3), the column in the result table is CHAR(6). The values in the result table that are derived from *DEPTNO* are padded on the right with blanks.

Rules for Numeric Columns

If you want to combine two numeric columns, A and B, the following rules apply:

- If column A or column B is floating-point, the result column is floating-point. If either column A or column B is double-precision floating-point, the result column is double-precision floating-point. If column A and column B are single-precision floating-point, the results column is single-precision floating-point. If either column A or column B is single-precision floating-point and the other is decimal, zoned decimal (NUMERIC), or binary integer, the result column is double-precision floating-point.
- If column A and column B are decimal, or one is decimal and the other binary integer or zoned decimal (NUMERIC), the results column is decimal.
- If column A and column B are zoned decimal (NUMERIC), or one is zoned decimal and the other binary integer, the results column is zoned decimal (NUMERIC).
- If column A or column B are large integer, the results column is large integer. If column A and column B are small integer, the results column is small integer.
- If column A and column B are nonzero scale binary and have different scales, or if one is nonzero scale binary and the other is zero scale binary integer, the results column is decimal.
- If column A and column B are nonzero scale binary and have the same scales, the results column is nonzero scale binary with the scale of A and B.

Rules for Character String Columns

If you want to combine two string columns, A and B, the following rules apply:

- Column A and column B must both be character string.
- If column A and column B are fixed length, the results column is fixed length. Otherwise, the results column is varying-length. The length attribute of the results column is the greater of the length attributes of column A and column B.

Specifying UNION ALL

If you want to keep duplicates in the result of a UNION, specify UNION ALL instead of just UNION.

Step 3. SQL combines two interim result tables:	Resulting in a result table that includes duplicates:

EXEC SQL
  DECLARE XMP6 CURSOR FOR
      SELECT ...
    UNION ALL
      SELECT ...
    ORDER BY 1
END-EXEC.
 ...
EXEC SQL
  FETCH XMP6
    INTO :EMP-NUMBER
END-EXEC.
	fetch EMP-NUMBER
	1 → 000060
	2 → 000150
	3 → 000160
	4 → 000170
	5 → 000170
	6 → 000180
	7 → 000180
	8 → 000190
	... ...

RSLS795-0

- The UNION ALL operation is associative, for example:

```
(SELECT PROJNO FROM USER1.TPROJ
UNION ALL
SELECT PROJNO FROM USER1.TPROJEC)
UNION ALL
SELECT PROJNO FROM USER1.TEMPRACT
```

  gives the same result as:

```
SELECT PROJNO FROM USER1.TPROJ
UNION ALL
(SELECT PROJNO FROM USER1.TPROJEC
UNION ALL
SELECT PROJNO FROM USER1.TEMPRACT)
```

- When you include the UNION ALL in the same SQL statement as a UNION operator, however, the result of the operation depends on the order of evaluation. Where there are no parentheses, evaluation is from left to right. Where parentheses are included, the parenthesized subselect is evaluated first, followed, from left to right, by the other parts of the statement.

Using a Cursor

Assume that SQL builds a result table[2] to hold all the rows retrieved by running the select-statement. SQL then uses a **cursor** to make rows from the result table available to your program. A cursor identifies the current row of the result table specified by a select-statement. When you use a cursor, your program can retrieve each row sequentially from the result table until end-of-data (SQLCODE = +100) is reached. The set of rows obtained as a result of running the select-statement can consist of zero, one, or many rows, depending on the number of rows that satisfy the search condition.

Note: The select-statement referred to in this section must be within a DECLARE CURSOR statement, and cannot include an INTO clause. The DECLARE CURSOR statement defines and names the cursor and specifies the set of rows to be retrieved with the embedded select-statement.

The result table of a cursor is processed much like a sequential data set. The cursor must be opened (with an OPEN statement) before any rows are retrieved. A FETCH statement is used to retrieve the cursor's current row. FETCH can be run repeatedly until all rows have been retrieved. When the end-of-data condition occurs, you should close the cursor with a CLOSE statement (similar to end-of-file processing).

Your program can have several cursors; each cursor requires its own:

DECLARE CURSOR statement to define the cursor
OPEN and CLOSE statements to open and close the cursor
FETCH statement to retrieve rows from the cursor's result table

Example of How to Use a Cursor

Suppose your program examines data about people in department D11. The data is kept in the USER1.TEMPL table. The following example shows the SQL statements you would include in a program to define and use a cursor. In this example, the cursor is used by the program to process a set of rows from the USER1.TEMPL table.

2 SQL implements a result table different ways, depending on the complexity of the select-statement. However, the concept is the same.

SQL Statement	Described in Section
EXEC SQL DECLARE THISEMP CURSOR FOR SELECT EMPNO, LASTNAME, DEPTNO, JOBCODE FROM USER1.TEMPL WHERE DEPTNO = 'D11' FOR UPDATE OF JOBCODE END-EXEC.	"Step 1: Define the Cursor" on page 3-22.
EXEC SQL OPEN THISEMP END-EXEC.	"Step 2: Open the Cursor" on page 3-23.
EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.	"Step 3: Specify What to Do when End-of-Data Is Reached" on page 3-23.
EXEC SQL FETCH THISEMP INTO :EMP-NUM, :NAME2, :DEPT, :JOB-CODE END-EXEC.	"Step 4: Retrieve a Row Using the Cursor" on page 3-23.
... for specific employees in department D11, update the JOBCODE value: EXEC SQL UPDATE USER1.TEMPL SET JOBCODE = :NEW-CODE WHERE CURRENT OF THISEMP END-EXEC. ... then print the row.	"Step 5a: Update the Current Row" on page 3-24.
... for other employees, delete the row: EXEC SQL DELETE FROM USER1.TEMPL WHERE CURRENT OF THISEMP END-EXEC.	"Step 5b: Delete the Current Row" on page 3-24.
Branch back to fetch and process the next row.	
CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.	"Step 6: Close the Cursor" on page 3-25.

Step 1: Define the Cursor

To define and identify a set of rows to be accessed with a cursor, issue a DECLARE CURSOR statement:

1. The DECLARE CURSOR statement names a cursor and specifies a select-statement. The select-statement defines a set of rows that, conceptually, makes up the result table. The statement looks like this:

   ```
   EXEC SQL
     DECLARE cursor-name CURSOR FOR
       SELECT column-1, column-2
         FROM table-name
         WHERE column-1 = search-condition
       FOR UPDATE OF column-2
   END-EXEC.
   ```

   The select-statement shown here is rather simple. However, you can code several other types of clauses in a select-statement within a DECLARE CURSOR statement.

2. If you intend to update any columns in any or all of the rows of the identified table (the table named in the FROM clause), include the FOR UPDATE OF clause; it names each column you intend to update. If you do not specify the names of columns you will later update, and you specify the ORDER BY clause, a negative SQLCODE is returned if an update is attempted. If you do not specify the FOR UPDATE OF clause and you do not specify the ORDER BY clause, you can update any of the columns of the specified table.

   You can update a column of the identified table even though it is not part of the result table. In this case, you do not need to name the column in the SELECT statement. When the cursor retrieves a row (using FETCH) that contains a column value you want to update, you can use UPDATE ... WHERE CURRENT OF to update the row.

   For example, assume that each row of the result table includes the *EMPNO*, *LASTNAME*, and *DEPTNO* columns from the USER1.TEMPL table. If you want to update the *JOBCODE* column (one of the columns in each row of the USER1.TEMPL table), the DECLARE CURSOR statement should include FOR UPDATE OF JOBCODE ... even though JOBCODE is omitted from the SELECT statement.

3. The result table is read-only if the SELECT statement includes the keyword DISTINCT, a UNION operator, a column function, a GROUP BY clause, or a HAVING clause. The result table is also read-only if the FROM clause of the SELECT statement identifies a read-only view or identifies more than one table or view (that is, if it is "joined" with another table). For details about the join technique, see "Joining Data from More Than One Table" on page 3-29.

Step 2: Open the Cursor

To tell SQL that you are ready to process the first row of the result table, issue the OPEN statement. When your program issues the OPEN statement, SQL processes the select-statement within the DECLARE CURSOR statement to identify a set of rows, called a result table[3], using the current value of any host variables specified in the select-statement. The OPEN statement looks like this:

```
EXEC SQL
  OPEN cursor-name
END-EXEC.
```

Step 3: Specify What to Do when End-of-Data Is Reached

To find out when no rows are left to process, test the SQLCODE field for a value of 100 (that is, end-of-data). This condition occurs when the FETCH statement has retrieved the last row in the result table and your program issues a subsequent FETCH. For example:

```
 ...
 IF SQLCODE =100 GO TO DATA-NOT-FOUND.
```

An alternative to this technique is to code the WHENEVER NOT FOUND clause. The WHENEVER NOT FOUND clause can result in a branch to another part of your program, where a CLOSE statement is issued. The WHENEVER NOT FOUND clause looks like this:

```
EXEC SQL
  WHENEVER NOT FOUND GO TO symbolic-address
END-EXEC.
```

Your program should anticipate an end-of-data condition whenever a cursor is used to fetch a row, and should be prepared to handle this situation when it occurs.

Step 4: Retrieve a Row Using the Cursor

To move the contents of a selected row into your program's host variables, use the FETCH statement. The SELECT statement within the DECLARE CURSOR statement identifies rows that contain the column values your program wants (that is, the result table is defined), but SQL does not retrieve any data for your application program until FETCH is issued.

When your program issues the FETCH statement, SQL uses the cursor to point to the next row in the result table, making it the **current row**. SQL then moves the current row's contents into your program's host variables (specified with the INTO clause). This sequence is repeated each time FETCH is issued, until you have processed all rows in the result table.

SQL maintains the position of the current row (that is, the cursor points to the current row) until the next FETCH statement for the cursor is issued. The UPDATE statement does not change the position of the current row within the result table, although the DELETE statement does.

[3] A result table can contain zero, one, or many rows, depending on the extent to which the search condition is satisfied.

The FETCH statement looks like this:

```
EXEC SQL
  FETCH cursor-name
    INTO :host variable-1[, :host variable-2] ...
END-EXEC.
```

Step 5a: Update the Current Row

When your program has retrieved the current row, you can update its data by using the UPDATE statement with the WHERE CURRENT OF clause. The WHERE CURRENT OF clause specifies a cursor that points to the row you want to update. The UPDATE ... WHERE CURRENT OF statement looks like this:

```
EXEC SQL
  UPDATE table-name
    SET column-1 = value [, column-2 = value] ...
    WHERE CURRENT OF cursor-name
END-EXEC.
```

When used with a cursor, the UPDATE statement:

- Updates only one row—the current row
- Identifies a cursor that points to the row to be updated
- Requires that the column updated be named previously in the FOR UPDATE OF clause of the DECLARE CURSOR statement, if an ORDER BY clause was also specified

After you have updated a row, the cursor's position remains on that row (that is, it points to the current row) until you issue a FETCH statement for the next row.

Step 5b: Delete the Current Row

When your program has retrieved the current row, you can delete the row by using the DELETE statement. To do this, you issue a DELETE statement designed for use with a cursor; the WHERE CURRENT OF clause specifies a cursor that points to the row you want to delete. The DELETE ... WHERE CURRENT OF statement looks like this:

```
EXEC SQL
  DELETE FROM table-name
    WHERE CURRENT OF cursor-name
END-EXEC.
```

When used with a cursor, the DELETE statement:

- Deletes only one row—the current row
- Uses the WHERE CURRENT OF clause to identify a cursor that points to the row to be deleted

After you have deleted a row, you cannot update or delete another row using that cursor until you issue a FETCH statement for the next row.

"The DELETE Statement" on page 3-4 shows you how to use the DELETE statement to delete all rows that meet a specific search condition. You can also use the FETCH and DELETE ... WHERE CURRENT OF statements when you want to obtain a copy of the row, examine it, then delete it.

Step 6: Close the Cursor

If you have finished processing the rows of the result table and you want to use the cursor again, issue a CLOSE statement to close the cursor:

```
EXEC SQL
  CLOSE cursor-name
END-EXEC.
```

If you have finished processing the rows of the result table and you do not want to use the cursor again, you can let the system automatically close the cursor when the first SQL program in the program stack ends.

Because an open cursor still holds locks on referred-to-tables or views, you should explicitly close any open cursors as soon as they are no longer needed.

A Unit of Recovery and Open Cursors

If your program completes a unit of recovery (that is, it either commits or rolls back the changes made so far), all open cursors are automatically closed by SQL, unless you have specified HOLD. You can reopen the cursor, but you will begin processing at the beginning of the result table.

If you wish to continue processing from the current cursor position after a COMMIT or ROLLBACK, you must specify COMMIT HOLD or ROLLBACK HOLD. When HOLD is specified, any open cursors are left open and keep their cursor position so that processing can resume. All record locks are still released. Since a FETCH statement acquires a lock on the row that it retrieves, you should not normally issue a FETCH statement before a COMMIT.

Preventing Duplicate Rows

When SQL evaluates a select-statement, several rows might qualify to be in the result table, depending on the number of rows that satisfy the select-statement's search condition. Some of the rows in the result table might be duplicates. You can specify that you do not want any duplicates by using the DISTINCT keyword, followed by the list of column names:

```
SELECT DISTINCT JOBCODE, SEX
 ...
```

DISTINCT means you want to select only unique rows. If a selected row duplicates another row in the result table, the duplicate row is ignored (it is not put into the result table). For example, suppose you want a list of employee job codes. You do not need to know which employee has what job code. Because it is probable that several people in a department have the same job code, you can use DISTINCT to ensure that the result table has only unique values.

The following example shows how to do this:

```
EXEC SQL
  DECLARE XMP2 CURSOR FOR
    SELECT DISTINCT JOBCODE
      FROM USER1.TEMPL
      WHERE DEPTNO = :JOB-DEPT
END-EXEC.
 ...
EXEC SQL
  FETCH XMP2
    INTO :JOBCODE
END-EXEC.
```

The result is four rows (in this example, JOB-DEPT is set to D11).

fetch	JOBCODE
1 →	55
2 →	54
3 →	53
4 →	52

RSLS796-0

If you do not include DISTINCT in a SELECT clause, you might find duplicate rows in your result, because SQL retrieves the *JOBCODE* column's value for each row that satisfies the search condition.

Performing Complex Search Conditions

The following section explains more advanced things you can do with search conditions.

Keywords for Use in Search Conditions

A search condition can contain any of the keywords BETWEEN...AND, IN, and LIKE.

Note: Literal values are shown in the following examples to keep the examples simple. However, you could just as easily code host variables instead. Remember to precede each host variable with a colon.

- **BETWEEN ... AND ...** is used to specify a search condition that is satisfied by any value that falls on or between two other values. For example, to find all employees who were hired in 1987, you could use this:

  ```
  ...
   WHERE HIREDATE BETWEEN '870101' AND '871231'
  ```

  The BETWEEN keyword is inclusive. A more complex, but explicit, search condition that produces the same result is:

  ```
  ...
   WHERE HIREDATE >= '870101' AND HIREDATE <= '871231'
  ```

- **IN** says you are interested in rows in which the value of the specified expression is among the values you have listed. For example, to find the names of all employees in departments A00, C01, and E21, you could specify:

```
...
WHERE DEPTNO IN ('A00', 'C01', 'E21')
```

- **LIKE** says you are interested in rows in which a column value is similar to the value you supply. When you use LIKE, the value on the left is always a column name of character type. The value on the right must be a character value: either a character string, a mixed character string, a host variable that contains character data, or the USER special register.

  When you use LIKE, SQL searches for a character string similar to the one you specify. The degree of similarity is determined by two special characters used in the string that you include in the search condition:

  _ An underscore character stands for any single character.

  % A percent sign stands for an unknown string of 0 or more characters. SQL expects the unknown string not to begin in the first position in the column unless the % precedes the string. In that case, the unknown string can begin anywhere in the column.

  Note: If you are operating on mixed data, the following distinction applies: an EBCDIC underscore character refers to one EBCDIC character. No such restriction applies to the percent sign; that is, a percent sign refers to any number of characters of any type.

  Use the underscore character or percent sign either when you do not know or do not care about all the characters of the column's value. For example, to find out which employees live in Minneapolis, you could specify:

```
...
WHERE ADDRESS LIKE '%MINNEAPOLIS%'
```

  In this case, you should be sure that MINNEAPOLIS was not part of a street address or part of another city name. SQL returns any row with the string MINNEAPOLIS in the ADDRESS column, no matter where the string occurs.

  In another example, to list the towns whose names begin with SAN_, you could specify:

```
...
WHERE TOWN LIKE 'SAN %'
```

Multiple Search Condition within a WHERE Clause

You have seen how to qualify a request using one search condition. You can qualify your request further by coding a search condition that includes several predicates. The search condition you specify can contain any of the comparison operators or the keywords BETWEEN, IN, and LIKE.

You can join any two predicates with the connectors AND and OR. In addition, you can use the NOT keyword to specify that the desired search condition is the negated value of the specified search condition. A WHERE clause can have as many predicates as you want.

- **AND** says that, for a row to qualify, the row must satisfy both predicates of the search condition. For example, to find out which employees in department D21 were hired after December 31, 1987, you would specify:

```
...
WHERE DEPTNO = 'D21' AND HIREDATE > '871231'
```

- **OR** says that, for a row to qualify, the row can satisfy the condition set by either or both predicates of the search condition. For example, to find out which employees are in either department C01 or D11, you could specify[4]:

```
...
WHERE DEPTNO = 'C01' OR DEPTNO = 'D11'
```

- **NOT** says that, to qualify, a row must not meet the criteria set by the search condition or predicate that follows the NOT. For example, to find all employees in department E11 except those with a job code lower than 55, you could specify:

```
...
WHERE DEPTNO = 'E11' AND NOT JOBCODE < 55
```

When SQL evaluates search conditions that contain these connectors, it does so in a specific order. SQL first evaluates the NOT clauses, next evaluates the AND clauses, and then the OR clauses.

You can change the order of evaluation by using parentheses. The search conditions enclosed in parentheses are evaluated first. For example, to select all employees in departments E11 and E21 who have job codes greater than 53, you could specify:

```
...
WHERE JOBCODE > 53 AND
  (DEPTNO = 'E11' OR DEPTNO = 'E21')
```

The parentheses determine the meaning of the search condition. In this example, you want all rows that have a:

DEPTNO value of E11 or E21, and
JOBCODE value greater than 53

If you did not use parentheses:

```
...
WHERE JOBCODE > 53 AND DEPTNO = 'E11'
  OR DEPTNO = 'E21'
```

your result is different. The selected rows are rows that have:

DEPTNO = E11 and JOBCODE > 53, or
DEPTNO = E21, regardless of the JOBCODE value

---

4 You could also use IN to specify this request: WHERE DEPTNO IN ('C01', 'D11').

Joining Data from More Than One Table

In the SELECT examples you have seen so far, the retrieved information has usually been in one table. Sometimes the information you want is not in one table only. To form a row of the result table, you might want to retrieve some column values from a row in one table and some columns from a row in another table. You can use a select-statement to retrieve and join column values from two or more tables into a row.

Joining data (as described in this section) differs from using the UNION keyword in the following ways:

With UNION keyword, the column values specified in one select-statement must have compatible data types with the column values in the other select-statement. Also, the set of rows selected from one table is added to the end of the set selected from the other table, and duplicate rows are eliminated.

With join, column values from one row of a table are combined with column values from another row of another (or the same) table to form a single row of data. SQL examines both tables specified in the join select-statement to retrieve data from as many rows as meet the search criteria specified in the WHERE clause.

For example, suppose you want to retrieve, for project MA2112, the employee numbers, names, activity codes, and amount of time spent on that project. In other words, you want the *EMPNO* and *LASTNAME* columns from the USER1.TEMPL table and the *ACTNO* and *EMPTIME* columns from the USER1.TEMPRACT table. To find this information, you need to join the two tables.

To do this, you list the two tables you are joining in the FROM clause of the SELECT statement. If the tables have any column names that are the same, you qualify those column names by prefixing them with the name of the table:

```
EXEC SQL
  DECLARE XMP7 CURSOR FOR
    SELECT USER1.TEMPL.EMPNO, LASTNAME, ACTNO, EMPTIME
      FROM USER1.TEMPL, USER1.TEMPRACT
      WHERE USER1.TEMPL.EMPNO = USER1.TEMPRACT.EMPNO
        AND USER1.TEMPRACT.PROJNO = 'MA2112'
END-EXEC.
 ...
EXEC SQL
  FETCH XMP7
    INTO :PERSON, :NAME3, :ACTIV, :TIME
END-EXEC.
```

To better understand what results from these SQL statements, imagine that SQL goes through the following process:

Step 1: SQL selects all rows in USER1.TEMPRACT with a PROJNO value of MA2112:	Which results in an interim result table:

`...`
`SELECT USER1.TEMPL.EMPNO, ...`
`ACTNO, EMPTIME`
`FROM ... USER1.TEMPRACT`
`WHERE ...`
`USER1.TEMPRACT.PROJNO = 'MA2112'`
	(from USER1.TEMPRACT)
	EMPNO ACTNO EMPTIME
	000170 70 100
	000190 70 100

Step 2: SQL selects all rows from the USER1.TEMPL table with a EMPNO value equal to the EMPNO values from step 1. From each row, SQL selects the row's EMPNO and LASTNAME values.	Which results in another interim result table:

`...`
`SELECT USER1.TEMPL.EMPNO, LASTNAME`
`FROM USER1.TEMPL, ...`
`WHERE USER1.TEMPL.EMPNO =`
`USER1.TEMPRACT.EMPNO ...`
`...`
	(from USER1.TEMPL)
	EMPNO LASTNAME
	000170 YOSHIMURA
	000190 WALKER

Step 3: SQL combines the results of the previous two steps to form a result table that can be retrieved a row at a time with a FETCH statement:

```
EXEC SQL
  DECLARE XMP7 CURSOR FOR
    SELECT USER1.TEMPL.EMPNO, LASTNAME, ACTNO, EMPTIME
      FROM USER1.TEMPL, USER1.TEMPRACT
        WHERE USER1.TEMPL.EMPNO = USER1.TEMPRACT.EMPNO
        AND USER1.TEMPRACT.PROJNO = 'MA2112'
END-EXEC.
...
EXEC SQL
  FETCH XMP7
    INTO :PERSON, :NAME3, :ACTIV, :TIME
END-EXEC.
```

The combined result table:

fetch	PERSON	NAME3	ACTIV	TIME
1 →	000170	YOSHIMURA	70	100
2 →	000190	WALKER	70	100

RSLS797-0

The WHERE Clause and Joining Tables

The WHERE clause establishes a condition for joining two or more tables; that is, it gives a relationship between the two tables.

When tables are joined, the WHERE clause can be used to:

- Concatenate, on a one-to-one basis, a row from table A with a corresponding row from table B
- Concatenate distinct groups of rows from table A to different rows of table B
- Concatenate groups of rows from table A to selected rows of table B

In the preceding example, the statement tells SQL that you want columns from rows in the USER1.TEMPL table concatenated with columns from rows in the USER1.TEMPRACT table that have the same EMPNO value and a PROJNO value of MA2112.

If you do not include a WHERE clause, *each row* of the USER1.TEMPL table is concatenated with *every row* of the USER1.TEMPRACT table to form a result table, even though the data may be unrelated. In other words, if you do not include a WHERE clause, the number of rows in the result table is the product of the number of rows of each joined table.

For example, note that the lack of a WHERE clause in the following SQL statement causes a concatenation of each row of table A to every row of table B:

```
EXEC SQL
 SELECT * FROM A, B
END-EXEC.
```

The results look like this:

Table A consists of:		Table B consists of:		The results of the SELECT are:			
Column 1	Column 2	Column 1	Column 2	VAR1	VAR2	VAR3	VAR4
A1	AA1	B1	BB1	A1	AA1	B1	BB1
A2	AA2	B2	BB2	A1	AA1	B2	BB2
A3	AA3			A2	AA2	B1	BB1
				A2	AA2	B2	BB2
				A3	AA3	B1	BB1
				A3	AA3	B2	BB2

RSLS798-0

Some of the uses you may have for joining two or more tables are:

- To create a view

  Note: The view created in this manner cannot be processed using UPDATE, DELETE, or INSERT statements.

- To select data
- To join a table or view to itself

Notes on the Join Technique

When you join two or more tables:

- If there are common column names, you must prefix each common name with the name of the table (or a correlation name). Column names that are unique do not need a prefix.
- If you do not list the column names you want, but instead use SELECT *, SQL returns rows that consist of all the columns of the first table, followed by all the columns of the second table, and so on.
- If the GROUP BY clause is used in a view's definition, you cannot join the view with any other table.
- You must be authorized to select rows from each table or view specified in the FROM clause.

Inserting Multiple Rows into a Table

You can use a subselect within an INSERT statement to insert zero, one, or more rows selected from the table or view you specify into another table. The rows you select cannot be from the same table you are inserting into. SQL will not process the INSERT statement if the tables are the same.

One use for this kind of INSERT statement is to move data into a table you have created for summary data. For example, suppose you want a table that shows each employee's time commitments to projects. You could create a table called EMPTIME with the columns *EMPNUMBER*, *PROJNUMBER*, *STARTDATE*, *ENDDATE*, and *TTIME*, and then use the following INSERT statement to fill the table:

```
EXEC SQL
  INSERT INTO USER1.EMPTIME
       (EMPNUMBER, PROJNUMBER, STARTDATE, ENDDATE)
    SELECT EMPNO, PROJNO, EMSTDATE, EMENDATE
      FROM USER1.TEMPRACT
END-EXEC.
```

The subselect embedded in the INSERT statement is no different from the subselect you use to retrieve data. With the exception of UNION and ORDER BY, you can use all the keywords, column functions, and techniques used to retrieve data. SQL inserts all the rows that meet the search conditions into the table you specify. Inserting rows from one table into another does not affect any existing rows in either the source table or the target table.

Notes on Multiple Row Insertion

You should keep in mind the following notes when inserting multiple rows into a table:

- The number of columns implicitly or explicitly listed in the INSERT statement must equal the number of columns listed in the subselect.
- The data in the columns you are selecting must be compatible with the columns you are inserting into.
- In the event the subselect statement embedded in the INSERT returns no rows, an SQLCODE of 100 is returned to alert you that no rows were inserted. If you successfully insert rows, the SQLERRD(3) field of the SQLCA will have an integer representing the number of rows SQL actually inserted.
- If SQL finds an error while running the INSERT statement, SQL stops the operation. If you specify COMMIT (*CHG) or COMMIT (*ALL), nothing is inserted into the table and a negative SQLCODE is returned. If you specify COMMIT(*NONE), any rows inserted prior to the error remain in the table.
- You can join two or more tables with a subselect in an INSERT statement. Loaded in this manner, the table can be operated on with UPDATE, DELETE, and INSERT statements, because the rows exist as physically stored rows in a table.
- When inserting new rows, you cannot specify an insertion sequence.

Inserting Default Values into Columns

At the time a table is created, individual columns within that table can be defined such that default values are inserted if no other values are specified on an INSERT statement.

To cause SQL to insert a default value into a column, omit that column's name and value from your INSERT statement. Lacking any explicit assignment, SQL will automatically insert the default value into that column.

Note: If a table column was defined as NOT NULL rather than NOT NULL WITH DEFAULT, you must explicitly provide a value for this column in any rows you INSERT. If you omit such specifications, SQL returns a negative SQLCODE.

Also note that, when you insert a row into a view, there is an additional consideration. If the view does not contain all the columns of the base table, SQL inserts default values into those columns of the base table that are not in the view.

Chapter 4. Common Concepts and Rules for Using SQL with Host Languages

This chapter describes some concepts and rules that are common to using SQL statements in a host language that involve:

- Using host variables in SQL statements
- Handling SQL error and return codes
- Handling exception conditions with the WHENEVER statement

Using Host Variables in SQL Statements

When your program retrieves data, the values are put into data items defined by your program and specified with the INTO clause of a SELECT or FETCH statement. The data items are called **host variables**.

A host variable is a field in your program that is referenced in an SQL statement, usually as the source or target for the value of a column. The host variable and column must be data-type compatible. Host variables may not be used to identify SQL objects such as tables or views.

A **host structure** is a group of host variables used as the source or target for a set of selected values (for example, the set of values for the columns of a row).

Note: By using a host variable instead of a literal value in an SQL statement, you give the application program the flexibility it needs to process different rows in a table or view.

For example, instead of coding an actual department number in a WHERE clause, you can use a host variable set to the department number you are currently interested in.

Host variables are commonly used in SQL statements in these ways:

1. **In a WHERE clause:** You can use a host variable to specify a value in the predicate of a search condition, or to replace a literal value in an expression. For example, if you have defined a field called EMPID that contains an employee number, you can retrieve the name of the employee whose number is 000110 with:

```
MOVE '000110' TO EMPID.
EXEC SQL
  SELECT LASTNAME
    INTO :PGM-LASTNAME
    FROM USER1.TEMPL
    WHERE EMPNO = :EMPID
END-EXEC.
```

2. **As a receiving area for column values (named in an INTO clause):** You can use a host variable to specify a program data area that is to contain the column values of a retrieved row. The INTO clause names one or more host variables that you want to contain column values returned by SQL. For example, suppose you are retrieving the *EMPNO*, *LASTNAME*, and *DEPTNO* column values from rows in the USER1.TEMPL table. You could define a host variable in your

program to hold each column, then name the host variables with an INTO clause. For example:

```
EXEC SQL
  SELECT EMPNO, LASTNAME, DEPTNO
    INTO :CBLEMPNO, :CBLNAME, :CBLDEPT
    FROM USER1.TEMPL
    WHERE EMPNO = :EMPID
END-EXEC.
```

In this example, the host variable CBLEMPNO receives the value from EMPNO, CBLNAME receives the value from LASTNAME, and CBLDEPT receives the value from DEPTNO.

3. **As a value in a SELECT clause:** When specifying a list of items in the SELECT clause, you are not restricted to the column names of tables and views. Your program can return a set of column values intermixed with host variable values and literal constants. For example:

```
MOVE '000220' TO PERSON.
EXEC SQL
  SELECT 'A', LASTNAME, SALARY, :RAISE,
      SALARY + :RAISE
    INTO :PROCESS, :PERSON-NAME, :EMP-SAL,
      :EMP-RAISE, :EMP-TTL
    FROM USER1.TEMPL
    WHERE EMPNO = :PERSON
END-EXEC.
```

The results are:

PROCESS	PERSON-NAME	EMP-SAL	EMP-RAISE	EMP-TTL
A	LUTZ	29840	4476	34316

4. **As a value in other clauses of an SQL statement:**

   The SET clause in an UPDATE statement
   The VALUES clause in an INSERT statement

   For more information on clauses, see the *SQL/400 Reference* manual.

Assignment Rules

SQL column values are sent to (or assigned to) host variables during the running of FETCH and SELECT INTO statements. SQL column values are set from (or assigned from) host variables during the running of INSERT and UPDATE statements. All assignment operations observe the following rules:

- Numbers and strings are not compatible:

  Numbers cannot be assigned to string columns or string host variables.
  Strings cannot be assigned to numeric columns or numeric host variables.

- All character strings are compatible; all numeric values are compatible. Conversions are performed by SQL whenever necessary.
- A null value cannot be assigned to a host variable that does not have an associated indicator variable.

Rules for Character String Assignment

Rules regarding character string assignment are:

- When a character string is put into a column, the length of the character string value must not be greater than the length attribute of the column.
- When a mixed character results column is assigned to a mixed column, the value of the mixed character results column must be a valid mixed character string.
- When the value of a result column is put into a host variable and the character string value of the result column is longer than the length attribute of the host variable, the string is truncated on the right by the necessary number of characters. If this occurs, SQLWARN0 and SQLWARN1 (in the SQLCA) are set to W. When using C/400, for a varying length null terminated character variable, if only the null terminator is truncated, then SQLWARN0 is set to W and SQLWARN1 is set to N.
- When the value of a results column is put into a fixed-length host variable or when the value of a host variable is put into a fixed-length CHAR result column and the length of the character string value is less than the length attribute of the target, the character string is padded on the right with the necessary number of blanks.
- When a mixed character results column is truncated because the length of the host variable into which it was being inserted was less than the length of the string, the shift-in character at the end of the string is preserved. The result is, therefore, still a valid mixed character string.

Rules for Numeric Assignment

Rules regarding numeric assignment are:

- **The whole part of a number may be altered when converting it to single-precision floating-point**. Because a single-precision floating-point field can only contain seven decimal digits, any whole part of a number that contains more than seven digits is altered due to rounding.
- **The whole part of a number is never truncated.** If necessary, the fractional part of a number is truncated. If the number, as converted, does not fit into the target host variable or column, a negative SQLCODE is returned.
- Whenever a **decimal, numeric, or binary number** is assigned to a decimal, numeric, or binary column or host variable, the number is converted, if necessary, to the precision and scale of the target. The necessary number of leading zeros is appended or deleted; in the fractional part of the number, the necessary number of trailing zeros is appended, or the necessary number of trailing digits is eliminated.
- When a **binary or floating-point number** is assigned to a decimal or numeric column or host variable, the number is first converted to a temporary decimal or numeric number and then converted, if necessary, to the precision and scale of the target.
  - When a **halfword binary integer** (SMALLINT) with 0 scale is converted to decimal or numeric, the temporary result has a precision of 5 and a scale of 0.
  - When a **fullword binary integer** (INTEGER) is converted to decimal or numeric, the temporary result has a precision of 11 and a scale of 0.

- When a **floating-point number** is converted to decimal or numeric, the temporary result has a precision of 31 and the maximum scale that allows the whole part of the number to be represented without loss of either significance or accuracy.

Indicator Variables

An **indicator variable** is a halfword integer variable used to indicate whether its associated host variable has been assigned a null value:

- If a host variable is used any place other than in an INTO clause, it must not have an associated indicator variable containing a negative value. You are responsible for ensuring that this indicator is not set to a negative value.
- If the value for the result column is null, SQL puts a negative value in the indicator variable.
- If you do not use an indicator variable and the result column is a null value, a negative SQLCODE is returned.
- If the value for the result column causes a numeric conversion error or an arithmetic expression error, SQL sets the indicator variable to −2.

You can also use an indicator variable to verify that a retrieved character string value has not been truncated. If truncation occurs, the indicator variable contains a positive integer that specifies the original length of the string.

When SQL returns a value from a result column, you can test the indicator variable. If the value of the indicator variable is less than zero, you know the value of the results column is null. When SQL returns a null value, nothing is put into the host variable used to contain the value of the result column (the value of the host variable is unchanged).

You specify an indicator variable (preceded by a colon) immediately after the host variable. For example:

```
EXEC SQL
  SELECT COUNT(*), AVG(SALARY)
    INTO :PLICNT, :PLISAL:INDNULL
    FROM USER1.TEMPL
    WHERE JOBCODE < 52
END-EXEC.
```

You can then test INDNULL to see if it contains a negative value. If it does, you know SQL returned a null value.

Used with Host Structures

You can also specify an **indicator structure** (defined as an array of halfword integer variables) to support a host structure. If the results column values returned to a host structure can be null, you can append an indicator structure name to the host structure name. This allows SQL to notify your program about each null value returned to a host variable in the host structure.

For example, in COBOL:

```
01  SAL-REC.
    10 MIN-SAL            PIC S9(6)V99 USAGE COMP-3.
    10 AVG-SAL            PIC S9(6)V99 USAGE COMP-3.
    10 MAX-SAL            PIC S9(6)V99 USAGE COMP-3.
77  SALIND                PIC S9999 USAGE COMP-4 OCCURS 3 TIMES.
01  EDUC-LEVEL            PIC S9999 COMP-4.
    ...
    MOVE 20 TO EDUC-LEVEL.
    ...
    EXEC SQL
      SELECT MIN(SALARY), AVG(SALARY), MAX(SALARY)
        INTO :SAL-REC:SALIND
        FROM USER1.TEMPL
        WHERE EDUCLVL>:EDUC-LEVEL
    END-EXEC.
```

In this example, SALIND is an array containing 3 values, each of which can be tested for a negative value. If, for example, SALIND(1) contains a negative value, then the corresponding host variable in the host structure (that is, MIN-SAL) is not changed for the selected row.

In the above example, SQL selects the column values of the row into a host structure. Therefore, you must use a corresponding structure for the indicator variables to determine which (if any) selected column values are null.

Handling SQL Error Return Codes

When an SQL statement is processed in your program, SQL places a return code in the SQLCODE field. The return code indicates the success or failure of the running of your statement. If SQL encounters an error while processing the statement, the return code in SQLCODE is a negative number. If SQL encounters an exception but valid condition while processing your statement, the return code is a positive number. If your SQL statement is processed without encountering an error or exception condition, the return code is zero.

Note: There are situations when a zero return code is returned to your program and the result might not be satisfactory. For example, if a value was truncated as a result of running your program, the SQLCODE returned to your program is zero. However, one of the SQL warning flags (SQLWARN1) indicates truncation.

Warning: If you do not test for negative SQL return codes or specify a WHENEVER SQLERROR statement, your program will continue to the next statement. Continuing to run after an error can produce unpredictable results.

Because the SQLCA is a valuable problem-diagnosis tool, it is a good idea to include in your application programs the instructions necessary to display some of the information contained in the SQLCA. Especially important are the following SQLCA fields:

SQLCODE — Return code.

SQLERRD(3) — The number of rows updated, inserted, or deleted by SQL.

SQLWARN0 — If set to W, at least one of the SQL warning flags (SQLWARN1 through SQLWARN7) is set.

For more information about SQLCA, see Appendix B. SQLCA and SQLDA Control Blocks in the *SQL/400 Reference* manual.

Handling Exception Conditions with the WHENEVER Statement

The WHENEVER statement causes SQL to check the SQLCA and continue processing your program, or branch to another area in your program if an error, exception, or warning exists as a result of running an SQL statement. An exception condition handling subroutine (part of your program) can then examine the SQLCODE field to take an action specific to the error or exception situation.

The WHENEVER statement allows you to specify what you want to do whenever a general condition is true. You can specify more than one WHENEVER statement for the same condition. When you do this, the first WHENEVER statement applies to all subsequent SQL statements in the source program until another WHENEVER statement is specified.

The WHENEVER statement looks like this:

```
EXEC SQL
  WHENEVER condition action
END-EXEC.
```

There are three conditions you can specify:

SQLWARNING
: Specify SQLWARNING to indicate what you want done when SQLWARN0 = W or SQLCODE contains a positive value other than 100.

  Note: SQLWARN0 could be set for several different reasons. For example, if the value of a column was truncated when it was moved into a host variable, your program might not regard this as an error.

SQLERROR
: Specify SQLERROR to indicate what you want done when an error code is returned as the result of an SQL statement (SQLCODE < 0).

NOT FOUND
: Specify NOT FOUND to indicate what you want done when an SQLCODE of +100 is returned because:

  - After a single-row SELECT is issued or after the first FETCH is issued for a cursor, the data the program specifies does not exist.
  - After a subsequent FETCH, no more rows satisfying the cursor select-statement are left to retrieve.
  - After an UPDATE, a DELETE, or an INSERT, no row meets the search condition.

You can also specify the action you want taken:

CONTINUE — This causes your program to continue to the next statement.

GO TO label — This causes your program to branch to an area in the program. The label for that area may be preceded with a colon. The WHENEVER ... GO TO statement:

- Must be a section name or an unqualified paragraph name in COBOL
- Is a label in PL/I and C
- Is the label of a TAG in RPG

For example, if you are retrieving rows using a cursor, you expect that SQL will eventually be unable to find another row when the FETCH statement is issued. To prepare for this situation, specify a WHENEVER NOT FOUND GO TO ... statement to cause SQL to branch to a place in the program where you issue a CLOSE statement in order to close the cursor properly.

Note: A WHENEVER statement affects all subsequent *source* SQL statements until another WHENEVER is encountered.

In other words, all SQL statements coded between two WHENEVER statements (or following the first, if there is only one) are governed by the first WHENEVER statement, regardless of the path the program takes.

Because of this, the WHENEVER statement *must precede* the first SQL statement it is to affect. If the WHENEVER *follows* the SQL statement, the branch is not taken on the basis of the value of the SQLCODE set by that SQL statement. However, if your program checks the SQLCODE directly, the check must be done after the SQL statement is run.

The WHENEVER statement does not provide a CALL to a subroutine option. For this reason, you might want to examine the SQLCODE value after each SQL statement is run and call a subroutine, rather than use a WHENEVER statement.

Chapter 5. Using SQL Statements in COBOL Programs

The AS/400 system supports more than one version of COBOL. The SQL/400 program only supports the COBOL/400 language. This chapter describes the unique application and coding requirements for embedding SQL statements in a COBOL/400 program. Requirements for host structures and host variables are defined. The handling of return codes is described.

A detailed sample COBOL program, showing how SQL statements can be used, is provided in Appendix C.

Application Requirements

To run SQL statements, your COBOL program must have an SQL communication area (SQLCA). There are two ways to get the SQLCA into your program. Use the INCLUDE SQLCA or code a COBOL data item with the name SQLCODE. The pre-compiler will provide an SQLCA when it finds a declaration for SQLCODE. The SQLCODE must be defined as:

```
PIC S9(n) COMP-4.
```

where *n* is a positive integer from 5 to 9.

SQL Communication Area (SQLCA)

The SQLCA is where SQL returns information about the results of running each SQL statement. To include the SQLCA declaration, you can use the INCLUDE statement:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
 ...
     EXEC SQL
     INCLUDE SQLCA
     END-EXEC.
```

You can specify INCLUDE SQLCA wherever a 77 level or a record description entry can be specified in the WORKING STORAGE section. (Other INCLUDE statements can be specified anywhere in the DATA DIVISION or the PROCEDURE DIVISION.)

© Copyright IBM Corp. 1988, 1989

When you use the INCLUDE statement, the SQL COBOL precompiler includes COBOL source statements for the SQLCA:

```
01 SQLCA.
   05 SQLCAID        PIC X(8).
   05 SQLCABC        PIC S9(9) COMP-4.
   05 SQLCODE        PIC S9(9) COMP-4.
   05 SQLERRM.
      49 SQLERRML    PIC S9(4) COMP-4.
      49 SQLERRMC    PIC X(70).
   05 SQLERRP        PIC X(8).
   05 SQLERRD        OCCURS 6 TIMES
                     PIC S9(9) COMP-4.
   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 SQLSTATE       PIC X(5).
```

SQLCODE is replaced with SQLCADE when a declare for SQLCODE is found in the program and the SQLCA is provided by the precompiler.

For more information on SQLCA, see the *SQL/400 Reference* manual.

Coding Requirements

You must put SQL statements (other than the INCLUDE statement) in the PROCEDURE DIVISION. The SQL statement can be preceded by a paragraph name if a COBOL statement in the same place could be preceded by a paragraph name.

When you issue SQL statements from a COBOL program, begin each statement with EXEC SQL. EXEC SQL must be specified within one line. The remainder of the statement may be specified on subsequent lines.

Each SQL statement must have an ending delimiter. In COBOL programs, end each SQL statement with END-EXEC. If several SQL statements are coded with no COBOL statements between them, the ending delimiter of each SQL statement must include an ending period (END-EXEC.). For a single SQL statement between two COBOL statements, an ending period is optional (END-EXEC. or END-EXEC). The ending period might not be appropriate in a structured program or where an SQL statement is embedded in a COBOL clause. An ending period ends the preceding COBOL statement (an IF ... THEN statement, for example).

For example, an UPDATE statement issued from a COBOL program would look like this:

```
EXEC SQL
UPDATE USER1.TDEPT
  SET MGRNO = :MGR-NUM
  WHERE DEPTNO = :INT-DEPT
END-EXEC.
```

Including Code: You cannot use COBOL COPY verbs to include SQL statements. You must use the SQL INCLUDE statement. INCLUDE statements cannot be nested.

Continuation for SQL Statements: The line continuation rules for SQL statements are the same as those for other COBOL statements, except that EXEC SQL must be specified within one line. If you continue a string constant from one line to the next, the first nonblank character in the next line must be a string delimiter. If you continue a delimited identifier from one line to the next, the first nonblank character in the next line must be the SQL escape character.

Comments: You can include COBOL comment lines (* in column 7) within SQL statements.

Note: COBOL debugging lines (D in column 7) and page eject lines (/ in column 7) are treated as comment lines by the SQL precompiler.

Margins: Code SQL statements in columns 12 through 72. If EXEC SQL starts before the specified margin (that is, before column 12), the SQL precompiler will not recognize the statement.

Sequence Numbers: The source statements generated by the SQL precompiler are generated with the same sequence number as the SQL statement.

Reserved Words: Do not use host variable names that begin with SQL. In addition, a list of SQL reserved words can be found in the *SQL/400 Reference*.

Dynamic SQL in a COBOL Program: Dynamic SQL is an advanced programming technique described in Chapter 9. With dynamic SQL, your program can develop and then run SQL statements while the program is running. A SELECT statement with a variable SELECT list (that is, a list of the data to be returned as part of the query) that runs dynamically requires an SQL descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT. Because the SQLDA uses pointer variables, which are not supported by COBOL, an INCLUDE SQLDA statement cannot be specified in a COBOL program. An SQLDA must be set up by a PL/I or C program and passed to the COBOL program in order to use it.

COBOL Compile-Time Options: The COBOL PROCESS statement can be used to specify the compile-time options for the COBOL compiler. Although the PROCESS statement will be recognized by the COBOL compiler when it is called to create the program by the precompiler; the SQL precompiler itself does not recognize the PROCESS statement. Therefore, options that affect the syntax of the COBOL source such as APOST and QUOTE should not be specified in the PROCESS statement. Instead *APOST and *QUOTE should be specified in the OPTION parameter of the CRTSQLCBL command.

Using COBOL Host Variables in SQL Statements

When your program retrieves data, the values are put into data items defined by your program and specified with the INTO clause of a SELECT INTO or FETCH statement. The data items are called **host variables**. See "Using Host Variables in SQL Statements" on page 4-1 for more information.

Host Structures

A **host structure** is a named set of host variables that is defined in your program's DATA DIVISION. Host structures have a maximum of two levels, even though the host structure might itself occur within a multilevel structure. An exception is the declaration of a varying-length character string, which requires another level that must be level 49.

A host structure name can be a group name whose subordinate levels name elementary data items. For example:

```
01 A
  02 B
    03 C1 PICTURE ...
    03 C2 PICTURE ...
```

In this example, B is the name of a host structure consisting of the elementary items C1 and C2.

When writing an SQL statement using a qualified host variable name (for example, to identify a field within a structure), use the name of the structure followed by a period and the name of the field (that is, PL/I style). For example, specify STRUCTURE.FIELD rather than FIELD OF STRUCTURE or FIELD IN STRUCTURE. However, PL/I style applies only to qualified names within SQL statements; you cannot use this technique for writing qualified names in COBOL statements.

After the host structure is defined, you can refer to it in an SQL statement instead of listing the several host variables (that is, the names of the data items that comprise the host structure).

For example, you can retrieve all column values from selected rows of the table USER1.TEMPL with:

```
01 PEMPL.
   10 EMPNO              PIC X(6).
   10 FIRSTNME.
      49 FIRSTNME-LEN    PIC S9(4) USAGE COMP-4.
      49 FIRSTNME-TEXT   PIC X(12).
   10 MIDINIT            PIC X(1).
   10 LASTNAME.
      49 LASTNAME-LEN    PIC S9(4) USAGE COMP-4.
      49 LASTNAME-TEXT   PIC X(15).
   10 DEPTNO             PIC X(3).
  ...
  MOVE '000220' TO EMPNO.
  ...
  EXEC SQL
    SELECT *
      INTO :PEMPL
      FROM USER1.TEMPL
      WHERE EMPNO = :EMPNO
  END-EXEC.
```

Notice that in the declaration of PEMPL, two varying-length string elements are included in the structure: FIRSTNME and LASTNAME.

Basic Requirements for Host Variables

This section describes the coding rules for host variables, assignment rules, declarations for host variables, and indicator variables and structures as they apply to COBOL/400 programs.

Coding Rules

Precede a host variable with a colon when it is used within an SQL statement. Do not precede a host variable with a colon when you use it outside of an SQL statement.

You can use any valid COBOL variable name for host variables. For example, the name can contain hyphens except as the first or last character in the name. The first character of a host variable must be a character other than a hyphen.

You cannot specify JUSTIFIED or BLANK WITH ZERO when coding a host variable.

You can specify OCCURS only when defining an indicator structure. You cannot specify OCCURS for any other type of host variable.

Assignment Rules

SQL column values are sent to (or assigned to) host variables during the running of FETCH and SELECT INTO statements. SQL column values are set from (or assigned from) host variables during the running of INSERT and UPDATE statements. All assignment operations observe the rules specified in the section "Assignment Rules" on page 4-2.

Allowable COBOL Declarations

SQL does not recognize every possible COBOL data description. If the COBOL data item you code is not consistent with those shown in the following table, the data item is not recognized as a host variable and might result in an error message when the program is precompiled. You can examine the data types in the cross-reference listing of the precompiler to note which variables have recognized data types.

The table shows the COBOL declarations that are allowed for use as host variables. All other types of COBOL declarations are rejected by SQL if encountered.

Note: Under *Explanatory Notes*, the length of string variables does not effect their recognition by the precompiler; the length limits apply to use, not declaration.

Table 5-1. COBOL Declarations Allowed for Use as Host Variables

SQL Data Type	COBOL Equivalent	Explanatory Notes
CHAR (character string)	01 identifier PIC X(n).	**n** is a positive integer. Anything other than Xs in the PICTURE clause makes the data item unacceptable as a host variable.
(varying-length character string)	01 identifier 49 identifier PIC S9(ni) COMP-4. 49 identifier PIC X(nc).	**ni** is a positive integer from 1 to 4. **nc** is a positive integer.
SMALLINT (halfword integer)	01 identifier PIC S9(n) COMP-4. or 01 identifier PIC S9(n)V9(m) COMP-4.	**n** is a positive integer from 1 to 4. **m** also is a positive integer; **n + m** cannot exceed 4. You can include a **V** to denote the decimal point. Anything other than a 9 and a V makes the data item unacceptable as a host variable. PIC S and COMP-4 are required.
INTEGER (fullword integer)	01 identifier PIC S9(n) COMP-4. or 01 identifier PIC S9(n)V9(m) COMP-4.	**n** is a positive integer from 5 to 9. **m** also is a positive integer; **n + m** can range from 5 to 9. You can include a **V** to denote the decimal point. Anything other than a 9 and a V makes the data item unacceptable as a host variable. PIC S and COMP-4 are required.
DECIMAL (decimal value)	01 identifier PIC S9(n)V COMP-3. or 01 identifier PIC S9(n)V9(m) COMP-3. or 01 identifier PIC S9(n)V9(m) COMP.	**n** and **m** are positive integers; **n + m** cannot exceed 18. You must include a **V** to denote the decimal point. Anything other than a 9 and a V makes the data item unacceptable as a host variable. COMP may be used in place of COMP-3. PIC S and COMP-3 or COMP are required.
NUMERIC (zoned decimal)	01 identifier PIC S9(n)V9(m). or 01 identifier PIC S9(n)V9(m) DISPLAY.	**n** and **m** are positive integers; **n + m** cannot exceed 18. You must include a **V** to denote the decimal point. Anything other than a 9 and a V makes the data item unacceptable as a host variable. PIC S is required.

The following COBOL abbreviations are acceptable when coding COBOL host variables:

PIC	PICTURE or PICTURE IS or PIC IS
COMP	COMPUTATIONAL or USAGE IS COMPUTATIONAL
USAGE	USAGE IS (an optional clause)
S9(4)	S9999
X(3)	XXX

Notes:

1. For data declarations of host variables not within a host structure, use level number 01 or 77. Host variables within a structure may be level 02 through 48. A leading zero is not required. Host structures have a maximum of two levels, except for level 49. Host structures may be used within levels 02 through 48.
2. A varying-length character string data item must have group level 01 through 48. The group must contain two elementary items with the level 49 and consists of:
   - Length item. The first elementary item must be a halfword integer variable (PICTURE S9(4) COMP-4. It represents the length of the character string.
   - Value item. The second elementary item must have the same description as a fixed-length character string (for example, PICTURE X(80), where 80 is the maximum length of the string). This item is used to contain the value of the character string. If you use the host variable to insert a character string into a column, SQL inserts only as many characters as are allowed by the length item.

   The data item and elementary items can have any acceptable COBOL name. However, you refer to this host variable in your SQL statements with its group name only.
3. Any level 77 data description entry except the length item of a varying-length character string may be followed by one or more REDEFINES or RENAMES entries. However, the names in these entries may not be used in SQL statements. Entries with the name FILLER are ignored.
4. The VALUE clause may be used to specify the initial contents of a data item.
5. Arrays (OCCURS clause) are not supported other than as indicator variables for structures.
6. COBOL host variables used in SQL statements must be type compatible with the columns with which they are to be used:
   - Numeric data types are compatible with each other.
   - Character data types are compatible with each other. SQL automatically converts a fixed-length character string to a varying-length string, or vice versa, when necessary.
7. Be careful of overflow. For example, if you retrieve an INTEGER column value into a PICTURE S9(4) host variable and the column value is larger than 32767 or smaller than −32768, you get an overflow error.
8. Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a PICTURE X(70) host variable, the rightmost 10 characters of the retrieved string are truncated.

Indicator Variables

An indicator variable is a halfword integer variable used to indicate whether its associated host variable has been assigned a null value. See "Indicator Variables" on page 4-4 for more information.

External Descriptions

SQL uses the COPY DD-format-name, COPY DD-ALL-FORMATS, COPY DDS-format-name, and COPY DDS-ALL-FORMATS to retrieve host variables from the file definitions. If the REPLACING option is specified, only complete name replacing is done. Identifier-1 is compared against the format name and the field name. If they are equal, identifier-2 is used as the new name.

Note: You cannot retrieve host variables from file definitions that have field names which are COBOL reserved words.

To retrieve the definition of the sample table TDEPT described in Appendix B, you can code the following:

```
01   TDEPT-STRUCTURE.
     COPY DDS-ALL-FORMATS OF TDEPT.
```

A host structure named TDEPT-STRUCTURE is defined with an 05 level field named TDEPT-RECORD that contains four 06 level fields named DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT. These field names can be used as host variables in SQL statements. For more information on the COBOL COPY verb, see the *COBOL/400 User's Guide*.

The WHENEVER Statement

The target for the GOTO clause in the SQL WHENEVER statement must be a section name or an unqualified paragraph name in the COBOL source.

Handling SQL Error Return Codes

When an SQL statement is processed in your program, SQL places a return code in the SQLCODE field. For more information on handling error return codes, see "Handling SQL Error Return Codes" on page 4-6. For a description of SQLCA, see the *SQL/400 Reference* manual.

Chapter 6. Using SQL Statements in RPG Programs

RPG/400 supports both RPG II and RPG III programs. SQL statements can only be used in RPG III programs. RPG II and AutoReport are **not** supported. All references to RPG in this manual apply to RPG III only.

This chapter describes the unique application and coding requirements for embedding SQL statements in an RPG program. Requirements for host structures and host variables are defined. The handling of return codes is also described.

A detailed sample RPG program, showing how SQL statements can be used, is provided in Appendix C.

Application Requirements

To run SQL statements, your RPG III program must have an SQL communication area (SQLCA).

SQL Communication Area (SQLCA)

The SQL precompiler automatically places the SQLCA data structure into the source prior to the first calculation specification. You should not code an INCLUDE SQLCA in your RPG program. If the program has an INCLUDE SQLCA specified, it is accepted and ignored. For more information on SQLCA, see the *SQL/400 Reference* manual.

© Copyright IBM Corp. 1988, 1989

The SQLCA, as defined for RPG:

```
ISQLCA       DS                                                       SQL
I*      SQL Communications area                                       SQL
I                                            1   8 SQLAID             SQL
I                                         B  9  120SQLABC             SQL
I                                         B 13  160SQLCOD             SQL
I                                         B 17  180SQLERL             SQL
I                                           19  88 SQLERM             SQL
I                                           89  96 SQLERP             SQL
I                                           97 120 SQLERR             SQL
I                                         B 97 1000SQLER1             SQL
I                                         B 101 1040SQLER2            SQL
I                                         B 105 1080SQLER3            SQL
I                                         B 109 1120SQLER4            SQL
I                                         B 113 1160SQLER5            SQL
I                                         B 117 1200SQLER6            SQL
I                                           121 127 SQLWRN            SQL
I                                           121 121 SQLWN0            SQL
I                                           122 122 SQLWN1            SQL
I                                           123 123 SQLWN2            SQL
I                                           124 124 SQLWN3            SQL
I                                           125 125 SQLWN4            SQL
I                                           126 126 SQLWN5            SQL
I                                           127 127 SQLWN6            SQL
I                                           128 128 SQLWN7            SQL
I                                           129 129 SQLWN8            SQL
I                                           130 130 SQLWN9            SQL
I                                           131 131 SQLWNA            SQL
I                                           132 136 SQLSTT            SQL
I*  End of SQLCA                                                      SQL
```

Note: Variable names in RPG are limited to 6 characters. The standard SQLCA names have been changed to a length of 6. RPG does not have a way of defining arrays in a data structure without also defining them in the extension specification. SQLERR is defined as character with SQLER1 through 6 used as the names of the elements.

Coding Requirements

SQL statements coded in RPG programs must be placed in the calculation section. This requires that a C be placed in position 6. SQL statements can be placed in detail calculations, in total calculations, or in an RPG subroutine. The SQL statements are run based on the logic of the RPG statements.

An SQL statement is delimited by /EXEC SQL and /END-EXEC.

EXEC SQL indicates the beginning of a SQL statement. It must occupy positions 8 through 16 of the source statement, preceded by a / in position 7. The SQL statement may start in position 17 and continue through position 74.

END-EXEC ends the SQL statement. It must occupy positions 8 through 16 of the source statement, preceded by a / in position 7. Positions 17 through 74 must be blank.

Including Code: You may use either the RPG /COPY statement or the SQL INCLUDE statement to include code into your source.

Continuation for SQL Statements: When additional records are needed to contain the SQL statement, positions 9 through 74 can be used. Position 7 must be a + (plus sign), and position 8 must be blank.

Comments: RPG comments can be embedded within the SQL statement by placing an * in position 7.

The following is an example of an SQL statement in RPG:

Note: Both uppercase and lowercase letters are acceptable in SQL statements.

```
C          SWITCH    IFEQ ON
C                    MOVE 'E01'  HOSTV 3
C/EXEC SQL insert into TDEPT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT)
C+ values('A00', 'COMPUTER SERVICE DIV', '000010', :HOSTV)
C/END-EXEC
C                    END
```

The SQL statement inside the delimiters can be coded free-form and is not restricted to uppercase notation. The following example is the previous example coded differently:

```
C          SWITCH    IFEQ ON
C                    MOVE 'E01'  HOSTV 3
C/EXEC SQL
C+     INSERT into TDEPT  (DEPTNO,
C+                                DEPTNAME,
C+                                  MGRNO,
C+                                     ADMRDEPT)
C*  this is a comment
C+        values ('A00',
C+                             'COMPUTER SERVICE DIV',
C+                               '000010',
C+                                  :hostv)
C/END-EXEC
C                     END
```

Sequence Numbers: The source statements generated by the SQL precompiler are generated with the same sequence number as the SQL statement.

Reserved Words: Do not use host variable names that begin with SQL. In addition, a list of SQL reserved words can be found in the *SQL/400 Reference*.

Dynamic SQL in a RPG Program: Dynamic SQL is an advanced programming technique described in Chapter 9. With dynamic SQL, your program can develop and then run SQL statements while the program is running. A SELECT statement with a variable SELECT list (that is, a list of the data to be returned as part of the query) that runs dynamically requires an SQL descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT. Because the SQLDA uses pointer variables which are not supported by RPG, an INCLUDE SQLDA statement cannot be specified in an RPG program. An SQLDA must be set up by a PL/I or C program and passed to the RPG program in order to use it.

Using RPG Host Variables in SQL Statements

When your program retrieves data, the values are put into RPG fields defined by your program and specified with the INTO clause of a SELECT INTO or FETCH statement. The RPG fields are called **host variables**. See "Using Host Variables in SQL Statements" on page 4-1 for more information.

Host Structures

The RPG data structure name can be used as a **host structure** name if subfields exist in the data structure. The use of the data structure name in an SQL statement implies the list of subfield names making up the data structure.

When subfields are not present for the data structure, then the data structure name is a host variable of character type. This allows character variables larger than 256, because data structures can be up to 9999.

In the following example, BIGCHR is an RPG data structure without subfields. SQL treats any references to BIGCHR as a character string with a length of 642.

```
IBIGCHR     DS                    642
```

In the next example, PEMPL is the name of the host structure consisting of the subfields EMPNO, FIRSTNME, MIDINIT, LASTNAME, and DEPTNO. The reference to PEMPL uses the subfields. For example, the first column of TEMPL is placed in *EMPNO*, the second column is placed in *FIRSTNME*, and so on.

```
IPEMPL      DS
I                                      01  06 EMPNO
I                                      07  18 FIRSTNME
I                                      19  19 MIDINT
I                                      20  34 LASTNAME
I                                      35  37 DEPTNO

...
C                    MOVE '000220'   EMPNO

...
C/EXEC SQL
C+ SELECT *  INTO :PEMPL
C+ FROM  USER1.TEMPL
C+ WHERE  EMPNO = :EMPNO
C/END-EXEC
```

When writing an SQL statement, references to subfields can be qualified. Use the name of the data structure, followed by a period and the name of the subfield. For example, PEMPL.MIDINT is the same as specifying only MIDINT.

Basic Requirements for Host Variables

This section describes the coding rules for host variables, assignment rules, declarations for host variables, and indicator variables and structures as they apply to RPG programs.

Coding Rules

Precede a host variable with a colon when it is used within an SQL statement. Do not precede a host variable with a colon when you use it outside of an SQL statement.

Any valid host variable name can be used in an SQL statement. You can use a host variable to represent a data value, but you cannot use it to represent a table name, view name, or column name.

Assignment Rules

RPG associates precision and scale with all numeric types. RPG defines numeric operations, assuming the data is in packed format. This means that operations involving binary variables include an implicit conversion to packed before the operation is performed (and back to binary, if necessary). Data is aligned to the implied decimal point when SQL operations are performed. All assignment operations observe the rules specified in the section "Assignment Rules" on page 4-2.

Allowable RPG Declarations

RPG has two types of data: character and numeric. Within the numeric type, there are four storage types: halfword binary, fullword binary, packed, and zoned. All variables defined in RPG can be used as host variables, except for the following:

- Multiple occurrence data structures
- Indicator field names (*INxx)
- Tables
- UDATE
- UDAY
- UMONTH
- UYEAR
- Look-ahead fields
- Named constants

Arrays cannot be used as host variables, but they can be used as indicator variables. Fields used as host variables are passed to SQL using the CALL/PARM functions of RPG. If a field cannot be used in the results field of the PARM, it cannot be used as a host variable.

The following table shows the relationship between SQL and RPG data types:

SQL Data Type	RPG Equivalent
CHAR (Character string)	Field defined without decimal places; maximum length, 9999.
SMALLINT (halfword integer)	Subfield of the data structure. B in position 43 of the subfield specification, storage length, 2. An entry must be made in position 52.

SQL Data Type	RPG Equivalent
INTEGER (fullword integer)	Subfield of the data structure. B in position 43 of the subfield specification, storage length, 4. An entry must be made in position 52.
DECIMAL (Decimal value)	Subfield of the data structure. P in position 43 and an entry in position 52 of the subfield specification. Or Defined as numeric and not a subfield of a data structure. Maximum precision of 30.
NUMERIC (Zoned decimal)	Subfield of the data structure. Blank in position 43. An entry must be made in position 52. Maximum precision of 30.

Indicator Variables

SQL requires the attribute of SMALLINT for data to be used as an indicator variable. This is accomplished in RPG by declaring the variable as a subfield in a data structure. The length for *to* and *from* must be 2. Position 43 must be B, and position 52 must be 0.

An indicator structure can be defined by declaring the variable as an array with an element length of 4,0 and declaring the array name as a subfield of a data structure with B in position 43.

The following example shows RPG field names used as indicator variables. The array AI has three elements defined as numeric 4,0. The entries for AI and IND as subfields declare their storage format of binary. AI can now be used as an indicator structure. IND can be used as an indicator variable.

```
E                    AI         3  4 0
.
.
.
I          DS
I                                   B  1  60AI
I                                   B  7  80IND
```

External Descriptions

The SQL precompiler processes the RPG source in much the same manner as the RPG compiler. This means that the precompiler processes the /COPY statement for definitions of host variables. Field definitions for externally described files are obtained and renamed, if different names are specified. The external definition form of the data structure can be used to obtain a copy of the column names to be used as host variables.

In the following example, the sample table TDEPT is used as a file in an RPG program. The SQL precompiler retrieves the field (column) definitions for TDEPT for use as host variables. The record name must be changed so that it is not the same as the file name. The field names must be changed because they are more than six characters in length.

```
FTDEPT   IP  E                    DISK
F            TDEPT                                KRENAMETDPTREC
ITDPTREC
I              DEPTNAME                           DEPTN
I              ADMRDEPT                           ADMRD
```

Note: Code an F-spec for a file in your RPG program only if you use RPG statements to do I/O operations to the file. If you use only SQL statements to do I/O operations to the file, include the external definition by using an external data structure.

In the following example, the sample table is specified as an external data structure. The SQL precompiler retrieves the field (column) definitions as subfields of the data structure. Subfield names can be used as host variable names, and the data structure name TDEPT can be used as a host structure name. The field names must be changed because they are greater than six characters.

```
ITDEPT   E   DS
I              DEPTNAME                           DEPTN
I              ADMRDEPT                           ADMRD
```

The WHENEVER Statement

The target for the GOTO clause in the SQL WHENEVER statement must be the label of an RPG TAG statement in the RPG source. The RPG scope rules for the GOTO/TAG must be observed. For more information on handling the WHENEVER statement, see “Handling Exception Conditions with the WHENEVER Statement” on page 4-7.

Handling SQL Error Return Codes

When an SQL statement is processed in your program, SQL places a return code in the SQLCOD field of the SQL communication area (SQLCA). The SQLCA allows SQL to communicate with your program. For more information on handling error return codes, see “Handling SQL Error Return Codes” on page 4-6. For a description of SQLCA, see the *SQL/400 Reference* manual.

Chapter 7. Using SQL Statements in PL/I Programs

This chapter describes the unique application and coding requirements for embedding SQL statements in an AS/400 PL/I program. Requirements for host structures and host variables are defined. The handling of return codes is described.

A detailed sample PL/I program, showing how SQL statements can be used, is provided in Appendix C.

Application Requirements

To run SQL statements, your PL/I program must have an SQL communication area (SQLCA). There are two ways to get the SQLCA into your program. Use the INCLUDE SQLCA or code a PL/I variable with the name SQLCODE. The precompiler will provide an SQLCA when it finds a declaration for SQLCODE. The SQLCODE must be defined as:

```
FIXED BINARY (31)
```

SQL Communication Area (SQLCA)

An AS/400 PL/I program that accesses SQL data must include an SQLCA. The SQLCA is the data area in which SQL returns information about the results of running each SQL statement. To include a PL/I declaration of the SQLCA, use the EXEC SQL INCLUDE statement:

```
EXEC SQL INCLUDE SQLCA;
```

When you use the EXEC SQL INCLUDE statement, SQL replaces that statement with a declaration of the SQLCA. The included PL/I source statements for the SQLCA are:

```
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 SQLSTATE      CHAR(5);
```

SQLCODE is replaced with SQLCADE when a declare for SQLCODE is found in the program and the SQLCA is provided by the precompiler.

For more information on SQLCA, see the *SQL/400 Reference* manual.

© Copyright IBM Corp. 1988, 1989

SQL Descriptor Area (SQLDA)

Dynamic SQL is an advanced programming technique and is described in Chapter 9. With dynamic SQL, your program can develop and then run SQL statements while the program is running.

A select-statement with a variable SELECT list (that is, a list of the data to be returned as part of the query) that runs dynamically requires an SQL descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT. The SQLDA is used to pass information about an SQL statement between SQL and your application. This information is used before an SQL statement is dynamically run. However, information about the results of running the statement is returned in the SQLCA. An SQLDA can be specified in a PL/I program by coding:

```
EXEC SQL INCLUDE SQLDA;
```

A PL/I program can also issue dynamic SQL statements using the EXECUTE IMMEDIATE statement, which does not require an SQLDA.

You may code the SQLDA 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 generated PL/I source statements for the SQLDA are:

```
DCL 1 SQLDA BASED(SQLDAPTR),
      2 SQLDAID       CHAR(8),
      2 SQLDABC       BIN FIXED(31),
      2 SQLN          BIN FIXED,
      2 SQLD          BIN FIXED,
      2 SQLVAR(99),
        3 SQLTYPE     BIN FIXED,
        3 SQLLEN      BIN FIXED,
        3 SQLRES      CHAR(12),
        3 SQLDATA     PTR,
        3 SQLIND      PTR,
        3 SQLNAME     CHAR(30) VAR;
DCL SQLDAPTR PTR;
```

Coding Requirements

You can code an SQL statement anywhere in a PL/I program.

When you issue SQL statements from a PL/I program, begin each statement with EXEC SQL. The EXEC SQL must be on one line. The remainder of the statement can appear on the next and subsequent lines.

Each SQL statement must have an ending delimiter. In PL/I programs, end each SQL statement with a semicolon (;). For example, an UPDATE statement issued from a PL/I program would look like this:

```
EXEC SQL
  UPDATE USER1.TDEPT
    SET MGRNO = :MGR_NUM
    WHERE DEPTNO = :INT_DEPT;
```

SQL statements, like PL/I statements, can have a label prefix.

Including Code: You cannot use the PL/I %INCLUDE directive to include source which contains SQL statements or host variable DCL statements. Use the SQL INCLUDE statement to include source which contains SQL statements and host variable DCL statements.

Continuation for SQL Statements: The line continuation rules for SQL statements are the same as those for other PL/I statements.

Comments: You can include PL/I comments in SQL statements wherever you can have a blank, except between the words EXEC and SQL.

Margins: Code SQL statements within the margins specified by the MARGINS parameter on the CRTSQLPLI command. If EXEC SQL does not start within the specified margins, the SQL precompiler will not recognize the SQL statement. For more information about the CRTSQLPLI command, see the section "Precompiler Commands" on page 10-7.

Reserved Words: Host variable names that begin with SQL are reserved for SQL. A list of additional SQL reserved words is located in the *SQL/400 Reference*.

Using PL/I Host Variables in SQL Statements

When your program retrieves data, the values are put into PL/I variables defined by your program and specified with the INTO clause of a SELECT INTO or FETCH statement. The PL/I variables are called **host variables**. See "Using Host Variables in SQL Statements" on page 4-1 for more information.

Host Structures

In PL/I programs, you can define a **host structure**, which is a named set of elementary PL/I variables. A host structure name can be a group name whose subordinate levels name elementary PL/I variables. For example:

```
DCL 1 A,
      2 B,
        3 C1 CHAR(...),
        3 C2 CHAR(...);
```

In this example, B is the name of a host structure consisting of the elementary items C1 and C2.

You can use the structure name as shorthand notation for a list of scalars, but only for a two-level structure. You can qualify a host variable with a structure name (for example, STRUCTURE.FIELD). Host structures are limited to two levels. (For example, in the above host structure example, the A cannot be referenced in SQL.) A structure cannot contain an intermediate level structure. In the previous example, A could not be used as a host variable or referenced in an SQL statement. A host structure for SQL data is two levels deep and can be thought of as a named set of host variables. After the host structure is defined, you can refer to it in an SQL statement instead of listing the several host variables (that is, the names of the host variables that make up the host structure).

For example, you can retrieve all column values from selected rows of the table USER1.TEMPL with:

```
DCL 1 PEMPL,
      5 EMPNO    CHAR(6),
      5 FIRSTNME CHAR(12) VAR,
      5 MIDINIT  CHAR(1),
      5 LASTNAME CHAR(15) VAR,
      5 DEPTNO CHAR(3);
 ...
EMPID = '000220';
 ...
  EXEC SQL
    SELECT *
      INTO :PEMPL
      FROM USER1.TEMPL
      WHERE EMPNO = :EMPID;
```

Basic Requirements for Host Variables

This section describes the coding rules for host variables, assignment rules, declarations for host variables, and indicator variables and structures as they apply to PL/I programs.

Coding Rules

Precede a host variable with a colon when it is used within an SQL statement. Do not precede a host variable with a colon when you use it outside of an SQL statement.

Any valid host variable name can be used in an SQL statement. You can use a host variable to represent a data value, but you cannot use it to represent a table name, view name, or column name.

Host variables must be scalars or structures of scalars. Host variables cannot be declared as arrays, although an array of indicator variables is allowed when the array is associated with a host structure.

You can use any valid PL/I variable name for host variables.

You can declare host variable attributes in any order acceptable to PL/I. For example, BIN FIXED(31), BINARY FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable. If several variables have exactly the same attributes, you can declare all of them in a single declare statement. For example:

```
DCL (X, Y, Z) BIN FIXED;
```

The names of host variables should be unique within the program, even if the host variables are in different blocks or procedures, unless you qualify them with a structure name to make them unique.

If host variables are used in an SQL statement, the SQL statement must be within the scope of the DCL statement in which the variable was declared.

Host variables can be STATIC, BASED, or AUTOMATIC storage class, or parameters.

Assignment Rules

SQL column values are sent to (or assigned to) host variables during the running of FETCH and SELECT INTO statements. SQL column values are set from (or assigned from) host variables during the running of INSERT and UPDATE statements. All assignment operations observe the rules specified in the section "Assignment Rules" on page 4-2.

Allowable PL/I Declarations

SQL will not recognize every possible PL/I data declaration. If the PL/I declaration you code is not consistent with those shown in the following table, the declaration is not recognized as a host variable and might result in an error message when the program is precompiled.

The following table shows the PL/I declarations that are allowed as host variables. All other types of PL/I declarations are ignored by SQL if encountered.

Table 7-1 (Page 1 of 2). PL/I Declarations Allowed for Use as Host Variables

SQL Data Type	PL/I Equivalent	Explanatory Notes
CHAR (character string)	DCL identifier CHAR(m);	**m** is a positive integer from 1 to 32767.
(Varying length character string)	DCL identifier CHAR(n) VAR;	**n** is a positive integer from 1 to 32765.
SMALLINT (halfword integer)	DCL identifier BIN FIXED; or DCL identifier FIXED BIN; or DCL identifier BIN FIXED(15); or DCL identifier FIXED BIN(15);	
INTEGER (fullword integer)	DCL identifier BIN FIXED(31); or DCL identifier FIXED BIN(31);	
DECIMAL (decimal value)	DCL identifier DEC FIXED(p); or DCL identifier FIXED DEC(p); or DCL identifier DEC FIXED(p,s); or DCL identifier FIXED DEC(p,s);	**p** (the precision) and **s** (the scale factor) are positive integers. **s** is 0 to 15 but cannot be larger than **p**. **p** is 1 to 15.
NUMERIC (zoned decimal)	DCL identifier PICTURE '999V99R';	Valid picture data items are 9, V, and R; R is required. The range of numeric values in the picture specification is expressed by 9 in digit positions, V for the decimal, and R for the sign. For example, imagine the expression `PICTURE '999V99R',` where the length equals 6, and the precision equals 3. V is optional, R is required.

Table 7-1 (Page 2 of 2). PL/I Declarations Allowed for Use as Host Variables

SQL Data Type	PL/I Equivalent	Explanatory Notes
REAL (single-precision floating-point)	DCL identifier BIN FLOAT(n); or DCL identifier FLOAT BIN(n); or DCL identifier DEC FLOAT(m); or DCL identifier FLOAT DEC(m);	**n** is a positive integer from 1 to 24, however, SQL recognizes it as FLOAT(24). **m** is a positive integer from 1 to 7, however, SQL recognizes it as FLOAT(24).
FLOAT(53) (double-precision floating-point)	DCL identifier BIN FLOAT(n); or DCL identifier FLOAT BIN(n); or DCL identifier DEC FLOAT(m); or DCL identifier FLOAT DEC(m);	**n** is a positive integer from 25 to 53, however, SQL recognizes it as FLOAT(53). **m** is a positive integer from 8 to 16, however, SQL recognizes it as FLOAT(53).

The following PL/I abbreviations are acceptable when coding PL/I host variables:

BIN	BINARY
CHAR	CHARACTER
DEC	DECIMAL
DCL	DECLARE
PIC	PICTURE
VAR	VARYING

Notes:

1. You cannot declare:

   - DECIMAL without FIXED or FLOAT
   - BINARY without FIXED or FLOAT
   - BIT
   - Implicit declarations
   - Block scoping rules
   - Arrays—other than for indicators

   Otherwise, any valid PL/I DECLARE statement can be coded (as long as the declaration is within the parameters noted in the preceding table). If BASED is coded, it must be followed by a PL/I element-locator-expression.

2. The INITIAL clause may be used to specify the initial contents of a host variable.

3. PL/I host variables used in SQL statements must be type compatible with the columns with which they are to be used:

   - Numeric data types are compatible with each other.

   - Character data types are compatible with each other. SQL automatically converts a fixed-length character string to a varying-length string, or vice versa, when necessary.

4. Be careful of overflow. For example, if you retrieve an INTEGER column value into a BIN FIXED(15) host variable and the column value is larger than 32767 or smaller than −32768, you get an overflow error.

5. Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHAR(70) host variable, the farthest right 10 characters of the retrieved string are truncated.

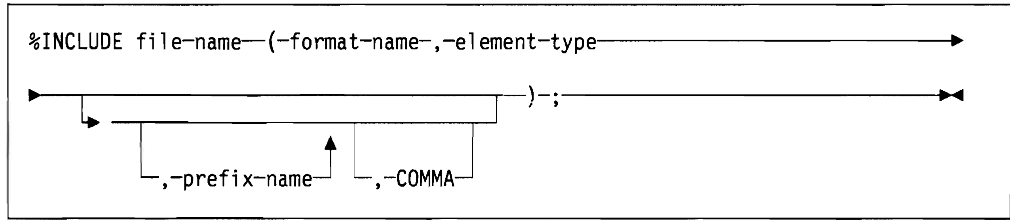
   Retrieving a floating-point, numeric, or decimal result column value into a BIN FIXED(31) or BIN FIXED(15) host variable removes any fractional value.

Indicator Variables

An indicator variable is a halfword integer variable used to indicate whether its associated host variable has been assigned a null value. See "Indicator Variables" on page 4-4 for more information.

Using the %INCLUDE Directive for External File Descriptions

You can use the PL/I %INCLUDE directive to include the definitions of externally described files in a source program. When used with SQL, only a particular format of the %INCLUDE directive is recognized by the SQL precompiler. That directive format must have the following three elements or parameter values, otherwise the precompiler ignores the directive. The required elements are *file name, format name,* and *element type*. There are two optional elements supported by the SQL precompiler: prefix name and COMMA. The syntax for the %INCLUDE directive and a description of the required and optional elements follows:



Element Description

A description of the required values follows:

file name
The name of the file that contains the record format to be included. You cannot name your file SYSLIB.

format name
The name of the record format to be included.

element type
The fields and indicators that are to be included. The valid types are:

INPUT
This type generates the record definition matching the input buffer. It includes fields that have a usage of INPUT or BOTH in the data description specification (DDS). OUTPUT fields are also included for subfiles.

OUTPUT
This type generates the record definition matching the output buffer. It includes fields that have a usage of OUTPUT or BOTH in DDS. For data base tables, it uses INPUT and BOTH.

KEY
This type includes fields specified as keys in DDS.

INDICATORS
This type is not supported by the AS/400 system SQL precompiler.

Note: If this element is present, the entire %INCLUDE directive is ignored by the precompiler.

RECORD
This type generates the record definition matching both the input and output buffers used by physical and logical files. It includes fields that have a usage of BOTH in DDS. INPUT fields are also included for logical files.

prefix name
A character string of 30 characters or less prefixing all generated names. The generated names must be valid and must be 31 characters or less.

COMMA
A value specifying that the last data element of the record or key structure does not end the structure. If COMMA is not specified (the default), the last data element ends the structure.

For more information on using the %INCLUDE directive, see the *PL/I User's Guide/Reference*.

Structure Definition

If the record format contains any of the fields of the element-type you specified, there are two factors that affect the names of the host variables that are generated: the *prefix* specified in the %INCLUDE directive, and the *alias* specified in the DDS source.

alias
If you use the ALIAS keyword in your DDS, the alias name you specify is the name the precompiler generates in your program. The field name specified in the DDS is ignored. By using the alias name, you can make full use of PL/Is 31-character name-length limit to increase the readability of your program.

prefix
If you code a prefix name in your %INCLUDE directive, the prefix you specify is attached to the field name or its alias supplied by the file. For example, if you write

```
%INCLUDE STOCK FILE(COUNT,RECORD,CURRENT_);
```

and, in the DDS, if the first field in the record is named

```
SALARY
```

the generated name is

```
CURRENT_SALARY
```

By using prefixes, you can generate meaningful names for different uses of the same record format inside a single program.

If one of the following conditions exists in the externally defined file, then the SQL precompiler generates a declaration of a host variable called DUMMYDCL:

- No fields are defined for the file.
- The element type is INPUT and no fields exist in the record format with usage of INPUT or BOTH.
- The element type is OUTPUT, the file is not a database file, and no fields exist in the record format with usage OUTPUT or BOTH.
- The element type is INDICATORS, but no separate indicators exist.

The DUMMYDCL host variable does not have a valid SQL data type and cannot be used. For example, a DUMMYDCL host variable is generated by a subfile control record format (which has no record fields and exists to define indicators and to communicate with the system).

A DUMMYDCL host variable is also generated if the record format you include has no fields of the element-type you specify. For example, if you specify INPUT for a record format that has only output fields, no fields are included, and a DUMMYDCL is generated.

The following table shows how each DDS data type is defined in a PL/I program:

DDS Data Type	Length	Decimal Position	SQL Equivalent
Indicator	1	0	Not usable as a host variable
A, H, J, E, O	1 to 32766	none	CHAR(n) where n = 1 to 32766
B	1 to 4	0	SMALLINT
B	5 to 9	0	INTEGER
B	1 to 4	1 to 4	CHAR(2)
B	5 to 9	1 to 9	CHAR(4)
P	1 to 15	0 to 15	DECIMAL(p,q) where: p = 1 to 15 q = 0 to 15
P	16 to 31	0 to 31	CHAR(n) where n = length ÷ 2 + 1
S	1 to 15	0 to 15	NUMERIC(p,q) where: p = 1 to 15 q = 0 to 15
S	16 to 31	0 to 31	CHAR(n) where n = 16 to 31
F	1 to 7	0 to 7	FLOAT(24)
F	8 to 15	0 to 15	FLOAT(53)

Structure Ending

The structure is ended normally by the last data element of the record or key structure. However, if in the %INCLUDE directive the COMMA element is specified, then the structure is not ended.

For more information about %INCLUDE structures, see the *PL/I User's Guide/Reference*.

%INCLUDE Example

To include the definition of the sample table TDEPT described in Appendix B, you can code:

```
DCL 1 TDEPT_STRUCTURE,
%INCLUDE TDEPT(TDEPT,RECORD);
```

In the above example, a host structure named TDEPT_STRUCTURE would be defined having four fields. The fields would be DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT.

The WHENEVER Statement

The target for the GOTO clause in the SQL WHENEVER statement must be a label in the PL/I source.

Handling SQL Error Return Codes

When an SQL statement is processed in your program, it places a return code in the SQLCODE field. For more information on handling error return codes, see "Handling SQL Error Return Codes" on page 4-6. For a description of SQLCA, see the *SQL/400 Reference* manual.

Chapter 8. Using SQL Statements in C Programs

This chapter describes the unique application and coding requirements for embedding SQL statements in a C/400 program. Requirements for host structures and host variables are defined. The handling of return codes is described.

A detailed sample C program, showing how SQL statements can be used, is provided in Appendix C.

Application Requirements

To run SQL statements, your C program must have an SQL communication area (SQLCA), as described below. There are two ways to get the SQLCA into your program. Use the INCLUDE SQLCA or code a C variable with the name SQLCODE. The precompiler will provide an SQLCA when it finds a declaration for SQLCODE. The SQLCODE must be defined as:

```
long int
```

SQL Communication Area (SQLCA)

A C/400 program must include one SQLCA within the scope of all SQL statements that can be run. The SQLCA is the data area that returns information about the results of running each SQL statement.

The SQLCA definition must be embedded before any SQL or C statements that can be run. The included C source statements for the SQLCA are:

```
#ifndef SQLCODE
struct sqlca {
              unsigned char sqlcaid[8];
              long          sqlcabc;
              long          sqlcode;
              short         sqlerrml;
              unsigned char sqlerrmc[70];
              unsigned char sqlerrp[8];
              long          sqlerrd[6];
              unsigned char sqlwarn[11];
              unsigned char sqlstate[5];
              };
#define SQLCODE sqlca.sqlcode,
#define SQLWARN0 sqlca.sqlwarn[0]
#define SQLWARN1 sqlca.sqlwarn[1]
#define SQLWARN2 sqlca.sqlwarn[2]
#define SQLWARN3 sqlca.sqlwarn[3]
#define SQLWARN4 sqlca.sqlwarn[4]
#define SQLWARN5 sqlca.sqlwarn[5]
#define SQLWARN6 sqlca.sqlwarn[6]
#define SQLWARN7 sqlca.sqlwarn[7]
#define SQLWARN8 sqlca.sqlwarn[8]
#define SQLWARN9 sqlca.sqlwarn[9]
#define SQLWARNA sqlca.sqlwarn[10]
#define SQLSTATE sqlca.sqlstate
#endif
struct sqlca sqlca;
```

© Copyright IBM Corp. 1988, 1989

When a declare for SQLCODE is found in the program and the SQLCA is provided by the precompiler, sqlcode is replaced with sqlcade. For more information on SQLCA, see the *SQL/400 Reference* manual.

SQL Descriptor Area (SQLDA)

Dynamic SQL is an advanced programming technique and is described in Chapter 9. With dynamic SQL, your program can develop and then run SQL statements while the program is running.

A select-statement with a variable SELECT list (that is, a list of the data to be returned as part of the query) that runs dynamically, requires an SQL descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT. The SQLDA is used to pass information about an SQL statement between SQL and your application. This information is used before an SQL statement is dynamically run. However, information about the results of running the statement is returned in the SQLCA. An SQLDA can be specified in a C program by coding:

```
EXEC SQL INCLUDE SQLDA;
```

The C declarations included for the SQLDA are:

```
#ifndef SQLDASIZE
struct sqlda {
              unsigned char sqldaid[8];
              long sqldabc;
              short sqln;
              short sqld;
              struct sqlvar {
                             short sqltype;
                             short sqllen;
                             unsigned char *sqldata;
                             short *sqlind;
                             struct sqlname {
                                             short length;
                                             unsigned char data[30];
                                            } sqlname;
                            } sqlvar[1];
              };
#define SQLDASIZE(n) (sizeof(struct sqlda) + (n-1)* sizeof(struct sqlvar))
#endif
```

One benefit from using the INCLUDE SQLDA SQL statement is that you also get the following macro definition:

```
#define SQLDASIZE(n) (sizeof(struct sqlda) + (n-1)* sizeof(struc sqlvar))
```

This macro makes it easy to allocate storage for an SQLDA with a specified number of SQLVAR elements. In the following example, the SQLDASIZE macro is used to allocate storage for an SQLDA with 20 SQLVAR elements.

```
#include <stdlib.h>
EXEC SQL INCLUDE SQLDA;

struct sqlda *mydaptr;
short numvars = 20;
  .
  .
mydaptr = (struct sqlda *) malloc(SQLDASIZE(numvars));
mydaptr->sqln = 20;
```

When you have declared an SQLDA as a pointer, you must dereference it when you use it as an SQL statement, just as you would for a host variable that was declared as a pointer. For example, if you declared a pointer to an SQLDA called mydaptr, you would use it in a PREPARE statement as:

```
EXEC SQL PREPARE mysname INTO :*mydaptr FROM :mysqlstring;
```

Coding Requirements

An SQL statement can be placed wherever a C statement that can be run can be placed.

The SQL statements begin with an EXEC SQL and end with a semicolon (;). The EXEC SQL must be all on one line. The rest of the SQL statement may be on more than one line. For example, an UPDATE statement coded in a C program would look like this:

```
EXEC SQL
     UPDATE USER1.TDEPT
     SET MGRNO = :mgrnum
     WHERE DEPTNO = :intdept;
```

Including Code: You cannot use the C #include preprocessor directive to include source which contains SQL statements or host variable declarations. Use the SQL INCLUDE statement to include source that contains SQL statements and host variable declarations.

Continuation for SQL Statements: SQL statements can be contained on one or more input lines. An SQL statement can be split wherever a blank can occur, except within a string constant. The backslash (\) is not supported.

Comments: Comments can appear within an SQL statement. Comments begin with the /* characters and end with */ characters, and span any number of lines. Comments cannot be nested.

Margins: You must code SQL statements within the margins specified by the MARGINS parameters on the CRTSQLC command.

Statement Labels: SQL statements that can be run can have statement labels associated with them, specified in the usual manner.

Reserved Words: Do not use host variable names that begin with SQL or sql. In addition, a list of SQL reserved words can be found in the *SQL/400 Reference*.

Nulls: C and SQL both use the word null, but for different meanings. The C language has a null character (NUL), a null pointer (NULL), and a null statement (just a semicolon). The C NUL is a single character that compares equal to 0. The C NULL is a special reserved pointer value that does not point to any valid data object. The SQL null value is a special value that is distinct from all non-null values and points out the absence of a (non-null) value.

Preprocessor Sequence: The SQL preprocessor must be run before the C preprocessor. No C preprocessor directives are permitted within SQL statements.

Using C Host Variables in SQL Statements

When your program retrieves data, the values are put into C variables defined by your program and specified with the INTO clause of a SELECT INTO or FETCH statement. The C variables are called host variables. See "Using Host Variables in SQL Statements" on page 4-1 for more information.

Host Structures

In C programs, you can define a **host structure**, which is a named set of elementary C variables. Host structures have a maximum of two levels, even though the host structure might itself occur within a multilevel structure. An exception is the declaration of a varying-length character string, that requires another structure.

A host structure name can be a group name whose subordinate levels name elementary C variables. For example:

```
struct {
        struct {
                char c1;
                char c2;
                } b_st;
        } a_st;
```

In this example, b_st is the name of a host structure consisting of the elementary items c1 and c2.

You can use the structure name as a shorthand notation for a list of scalars, but only for a two-level structure. You can qualify a host variable with a structure name (for example, structure.field). Host structures are limited to two levels. (For example, in the above host structure example, the a_st cannot be referenced in SQL.) A structure cannot contain an intermediate level structure. In the previous example, a_st could not be used as a host variable or referenced in an SQL statement. A host structure for SQL data is two levels deep and can be thought of as a named set of host variables. After the host structure is defined, you can refer to it in an SQL statement instead of listing the several host variables (that is, the names of the host variables that make up the host structure).

For example, you can retrieve all column values from selected rows of the table USER1.TEMPL with:

```
struct { char empno[7];
         struct         { short int firstname-len;
                          char firstname-text[12];
                        } firstname;
         char midint[1];
         struct         { short int lastname-len;
                          char lastname-text[15];
                        } lastname;
         char deptno[4];
       } pempl;
  .....
strcpy("000220",pempl.empno);
 .....
exec sql
   select *
     into :pempl
     from user1.templ
     where empno=:pempl.empno;
```

Notice that in the declaration of pemp1, two varying-length string elements are included in the structure: firstname and lastname.

Basic Requirements of Host Variables

This section describes the coding rules for host variables, indicator variables, and structures as they apply to C programs.

Coding Rules

Any valid host variable name, with a maximum of 31 characters can be used in an SQL statement. A host variable may be used to represent a data value, but cannot be used to represent a table name, view name, or column name.

When a host variable occurs within an SQL statement, the host variable must be preceded by a colon (:) to tell SQL that the variable is not a column name.

You must explicitly declare all host variables used in SQL statements. The declarations must appear in the program before the SQL or C statements that refer to the variable.

Host variables must be scalar variables; they cannot be elements of vectors or arrays (subscripted variables) except for character arrays used to hold strings.

Character host variables must include an explicit constant length. The declaration must not use an expression to define the length of the character variable.

With respect to SQL statements, all host variables have global scope regardless of where they are actually declared. Thus, although C allows you to use the same name for variables in different scopes, you can only use a name once for a host variable.

Assignment Rules

SQL column values are sent to (or assigned to) host variables during the running of FETCH and SELECT INTO statements. SQL column values are set from (or assigned from) host variables during the running of INSERT and UPDATE statements. All assignment operations observe the rules specified in the section "Assignment Rules" on page 4-2.

Allowable C Declarations

SQL will not recognize every possible C variable. If the C variable you code is not consistent with those shown in the following table, the variable is not recognized as a host variable and cannot be used with an SQL statement.

The following table shows the C variables that are allowed for use as host variables. All other types of C variables will be rejected.

Table 8-1. C Declarations Allowed for Use as Host Variables

SQL Data Type	C Equivalent	Explanatory Notes
CHAR(1) (character string)	char identifier;	A single character. C does not provide a fixed-length character string longer than one character.
No exact equivalent (character string length n)	char identifier[m];	Character array used to hold NUL terminated strings (m=n+1, n<=32766). The first NUL in the array ends the string.
(Varying-length character string)	struct { short len; char s[n]; } identifier;	Structure to emulate varying-length string.
SMALLINT (halfword integer)	short int identifier; or short identifier;	16 bit, signed integer.
INTEGER (fullword integer)	long int identifier; or long identifier;	32 bit, signed integer. No decimal point is allowed.
DECIMAL (decimal value)	no exact equivalent	Because C does not support decimal numbers, code a decimal column as a float or double variable or as an integer.
REAL (single-precision floating-point)	float identifier;	Floating point.
FLOAT (double-precision floating-point)	double identifier;	Floating point.
NUMERIC (zoned decimal)	no exact equivalent	Because C does not support zoned numbers, code a numeric column as a float or double variable or as an integer.

C supports some data types and storage classes with no SQL equivalents. For example:

typedef
register storage class
int type
unsigned integers
long double type
incomplete types

void type
enum type
union type
bitfield
array type attribute (except for char and indicator arrays)
function type attribute
const
signed
volatile

In most cases, C statements can be used to convert between these data types and the data types allowed by SQL.

Notes:

1. Storage class.

   The C storage class may be auto, extern, or static. Host variables cannot be register class. Host variables cannot be typedef class.

2. Character data type.

   The character data type in C is a NUL-terminated character string whose length is specified as n in the declaration above. The NUL is included in the length. The equivalent SQL data type is CHARACTER whose length is n-1 and which is not NUL-terminated. The SQLTYPE for the C NUL-delimited string is 460. It cannot be used for data when the data includes NUL. Use the varying-length character strings if the data can include NUL.

3. Varying-length character data type.

   The varying-length character string data item is supported in C for strings of arbitrary (binary or character) data. Because binary data can contain any combination, varying-length character strings use a length field and no string terminator. It is important to note that the C string manipulation functions cannot handle this data type, since the strings can contain NUL characters and are not null terminated, contrary to what those functions expect.

   ```
   struct { short length;
            char data[datalen];
          } identifier;
   ```

4. Apostrophes and quotes have different meanings in C and SQL. C uses quotes to delimit string constants and apostrophes to delimit character constants. SQL does not have this distinction, but quotes are used for delimited identifiers and apostrophes are used to delimit character string constants. Character data in SQL is distinct from integer data.

5. C host variables used in SQL statements must be type-compatible with the columns with which they are to be used.

   - Numeric data types are compatible with each other: a SMALLINT, INTEGER, DECIMAL, NUMERIC, or FLOAT column is compatible with a C host variable defined as short int, long int, float, or double.
   - Character data types (CHAR and varying-length string) are compatible with each other.

6. Be careful of overflow. For example, if you retrieve an INTEGER column value into a short int host variable and the column value is larger than 32767, you will get an overflow error.

7. Be careful to avoid truncation, ensure the host variable you declare can contain the data and a NUL terminator, if applicable.

   Retrieving a floating point, numeric, or decimal result column value into a short or long identifier host variable removes any fractional value.

Supported Pointer Data Types

You can also declare host variables that are pointers to the supported C data types, with the following restrictions:

- If a host variable is declared as a pointer, then that host variable must be declared with asterisks followed by a host variable. The following examples are all valid:

  ```
  short *mynum;                /* Ptr to an integer                  */
  long **mynumptr;             /* Ptr to a ptr to a long integer     */
  char *mychar;                /* Ptr to a single character          */
  char(*mychara)[20]           /* Ptr to a char array of 20 bytes    */
  struct {                     /* Ptr to a variable char array of 30 */
     short mylen;              /*     bytes.                         */
     char mydata[30];
     } *myvarchar;
  ```

  Note: Parentheses are only allowed when declaring a pointer to a null terminated character array, in which case they are required. If the parentheses were not used, you would be declaring an array of pointers rather than the desired pointer to an array. For example:

  ```
  char (*a)[10];         /* pointer to a null-terminated char array */
  char *a[10];           /* pointer to an array of pointers         */
  ```

- If a host variable is declared as a pointer, then no other host variable may be declared with that same name within the same source file. For example, the second declaration below would be invalid:

  ```
  char *mychar;               /* This declaration is valid          */
  char mychar;                /* But this one is invalid            */
  ```

- When a host variable is referenced within an SQL statement, that host variable must be referenced exactly as declared, with the exception of pointers to null-terminated character arrays. For example, the following declaration required parentheses:

  ```
  char (*mychara)[20];        /* ptr to char array of 20 bytes       */
  ```

  However, the parentheses are not allowed when the host variable is referenced in an SQL statement, such as a SELECT:

  ```
  EXEC SQL SELECT name INTO  :*mychara FROM mytable;
  ```

- Only the asterisk may be used as an operator over a host variable name.

- The maximum length of a host variable name is affected by the number of asterisks specified, as these asterisks are considered part of the name.

- Pointers to structures are not usable as host variables except for variable character structures. Also, pointer fields in structures are not usable as host variables.

Indicator Variables

An indicator variable is a short integer variable. You can also specify an indicator structure (defined as an array of halfword integer variables) to support a host structure.

The WHENEVER Statement

The target for the GOTO clause in the SQL WHENEVER statement must be a label in the C source.

Handling SQL Error Return Codes

When an SQL statement is processed in your program, it places a return code in the SQLCODE field. For more information on handling error return codes, see "Handling SQL Error Return Codes" on page 4-6. For a description of SQLCA, see the *SQL/400 Reference* manual.

Chapter 9. Dynamic SQL Applications

Dynamic SQL allows an application to define and run SQL statements at program run time. An application that provides for dynamic SQL accepts as input (or builds) an SQL statement in the form of a character string. The application does not need to know what type of SQL statement it will run. The application:

- Builds or accepts as input an SQL statement
- Prepares the SQL statement for running
- Runs the statement
- Handles SQL return codes

Interactive SQL (described in Chapter 11) is an example of a dynamic SQL program. SQL statements are processed and run dynamically by interactive SQL.

Note: The run-time overhead is greater for statements processed using dynamic SQL than for static SQL statements. The additional process is similar to that required for precompiling, binding, and then running a program, instead of only running it. Therefore, only applications requiring the flexibility of dynamic SQL should use it. Other applications should access data from the database using normal (static) SQL statements.

Some dynamic SQL statements require use of address variables. COBOL or RPG programs require the aid of PL/I or C programs to manage the address variables.

The examples in this chapter are PL/I examples. The following table shows all the statements supported by the SQL/400 program and indicates if they can be used in a dynamic application:

© Copyright IBM Corp. 1988, 1989

Note: In the following table, the numbers in the *Dynamic SQL* column correspond to the notes on the next page.

Table 9-1. List of SQL Statements Allowed in Dynamic Applications

SQL Statement	Static SQL	Dynamic SQL
BEGIN DECLARE SECTION	Y	N
CLOSE	Y	N
COMMENT ON	Y	Y
COMMIT	Y	Y
CREATE COLLECTION	Y	Y
CREATE INDEX	Y	Y
CREATE TABLE	Y	Y
CREATE VIEW	Y	Y
DECLARE CURSOR	Y	See Note 4.
DECLARE STATEMENT	Y	N
DELETE	Y	Y
DESCRIBE	Y	See Note 6.
DROP	Y	Y
END DECLARE SECTION	Y	N
EXECUTE	Y	See Note 1.
EXECUTE IMMEDIATE	Y	See Note 2.
FETCH	Y	N
GRANT	Y	Y
INCLUDE	Y	N
INSERT	Y	Y
LABEL ON	Y	Y
LOCK TABLE	Y	Y
OPEN	Y	N
PREPARE	Y	See Note 3.
REVOKE	Y	Y
ROLLBACK	Y	Y
SELECT	Y	See Note 5.
UPDATE	Y	Y
WHENEVER	Y	N

Notes:

1. Cannot be prepared, but used to run prepared SQL statements. The SQL statement must be previously prepared by the PREPARE statement prior to using the EXECUTE statement. See example for PREPARE under "Using the PREPARE and EXECUTE Statements" on page 9-4.
2. Cannot be prepared, but used with dynamic statement strings that do not have any ? parameter markers. The EXECUTE IMMEDIATE statement causes the statement strings to be prepared and run dynamically at program run time. See example for EXECUTE IMMEDIATE under "Processing NonSelect-Statements" on page 9-3.
3. Cannot be prepared, but used to parse, optimize, and set up dynamic SELECT statements prior to running. See example for PREPARE under "Processing NonSelect-Statements" on page 9-3.
4. Cannot be prepared, but used to define the cursor for the associated dynamic select-statement prior to running.
5. Cannot be used with EXECUTE or EXECUTE IMMEDIATE, but can be prepared and used with OPEN CURSOR. A SELECT INTO statement cannot be prepared or used in EXECUTE IMMEDIATE.
6. Cannot be prepared, but used to return a description of a prepared statement.

Designing and Running a Dynamic SQL Application

To issue a dynamic SQL statement, you must use the statement with either an EXECUTE statement or an EXECUTE IMMEDIATE statement, because dynamic SQL statements are not prepared at precompile time and therefore must be prepared at run time. The EXECUTE IMMEDIATE statement causes the SQL statement to be prepared and run dynamically at program run time.

There are two basic types of dynamic SQL statements: select-statements and nonselect-statements. Nonselect-statements include such statements as DELETE, INSERT, and UPDATE.

Processing NonSelect-Statements

To build a dynamic SQL nonselect-statement:

1. Verify that the SQL statement you want to build is one that can be run dynamically (see Table 9-1 on page 9-2).
2. Build the SQL statement. (Use Interactive SQL for an easy way to build, verify, and run your SQL statement. See Chapter 11 for more information.)

To run a dynamic SQL nonselect-statement:

1. Run the SQL statement using EXECUTE IMMEDIATE, or PREPARE the SQL statement, then EXECUTE the prepared statement.
2. Handle any SQL return codes that might result.

The following is an example of an application running a dynamic SQL nonselect-statement (stmtstrg):

```
EXEC SQL
  EXECUTE IMMEDIATE :stmtstrg;
```

Note: The SQL statement is normally a host variable. In PL/I, it can also be a string expression.

Using the PREPARE and EXECUTE Statements

If nonselect-statements contain no parameter markers, they can be run dynamically using the EXECUTE IMMEDIATE statement. However, if the nonselect-statements have parameter markers, they must be run using PREPARE and EXECUTE.

The PREPARE statement prepares the nonselect-statement (for example, the DELETE statement) and gives it a name of your choosing. In this instance, let us call it S1. After the statement has been prepared, it can be run many times within the same program, using different values for the parameter markers. The following example is of a prepared statement being run multiple times:

```
DSTRING = 'DELETE FROM USER1.TEMPL WHERE EMPNO = ?';

/*The ? is a parameter marker which denotes
  that this value is a host variable that is
  to be substituted each time the statement is run.*/

EXEC SQL PREPARE S1 FROM :DSTRING;

/*DSTRING is the delete statement that the PREPARE statement is
  naming S1.*/

 DO UNTIL (EMP =0);
/*The application program reads a value for EMP from the
  display station.*/
   EXEC SQL
      EXECUTE S1 USING :EMP;


 END;
```

In routines similar to the example above, you must know the number of parameter markers and their data types, because the host variables that provide the input data are declared when the program is being written.

Note: All SQL statements that have been prepared are destroyed when COMMIT or ROLLBACK is run, unless the HOLD is specified. For more information, see descriptions of COMMIT and ROLLBACK in the *SQL/400 Reference* manual.

Processing Select-Statements and Using SQLDA

There are two basic types of select-statements: **fixed-list** and **varying-list**.

To process a fixed-list select-statement, an SQLDA is not necessary.

To process a varying-list select-statement, you must first declare an SQLDA structure. SQLDA is a control block used to pass host variable input values from an application program to SQL and to receive output values from SQL. In addition, information about SELECT list expressions can be returned in a PREPARE or DESCRIBE statement.

Fixed-List Select-Statements

In dynamic SQL, fixed-list select-statements are those statements designed to retrieve data of a predictable number and type. When using these statements, you can anticipate and define host variables to accommodate the retrieved data, so that an SQLDA is not necessary. Each successive FETCH returns the same number of values as the last, and these values have the same data formats as those returned for the last FETCH. You can specify host variables the same as you would for any SQL application.

You can use fixed-list dynamic select-statements with any SQL-supported application program.

To run fixed-list select-statements dynamically, your application must:

1. Place the input SQL statement into a host variable.
2. Issue a PREPARE statement to validate the dynamic SQL statement and put it into a form that can be run.
3. Declare a cursor for the statement name.
4. Open the cursor.
5. FETCH a row into a fixed list of variables (rather than into a descriptor area, as you would if you were using a varying-list select-statement, described in the following section, "Varying-List Select-Statements").
6. When end of data occurs, close the cursor.
7. Handle any SQL return codes that may result.

For example,

```
DSTRING = 'SELECT EMP, EMPNAME
  FROM USER1.TEMPL WHERE EMP> ?';

EXEC SQL
  PREPARE S2 FROM :DSTRING;

EXEC SQL
  DECLARE C2 CURSOR FOR S2;

EXEC SQL
  OPEN C2 USING :EMP;

DO WHILE (SQLCODE = 0);

EXEC SQL
  FETCH C2 INTO :EMP, :EMPNAME;

END;

EXEC SQL
  CLOSE C2;
```

Note: Remember that because the select-statement, in this case, always returns the same number and type of data items as previously run fixed-list select-statements, you do not have to use the SQL descriptor area (SQLDA). The preceding example can be run with COBOL/400, AS/400 PL/I, RPG III (part of RPG/400), and C/400.

Varying-List Select-Statements

In dynamic SQL, varying-list select-statements are ones for which the number and format of result columns to be returned are not predictable; that is, you do not know how many variables you need, or what the data types are. Therefore, you cannot define host variables in advance to accommodate the result columns returned.

If your application accepts varying-list SELECT statements, your program has to:

1. Place the input SQL statement into a host variable.
2. Issue a PREPARE statement to validate the dynamic SQL statement and put it into a form that can be run.
3. Declare a cursor for the statement name.
4. Issue a DESCRIBE statement to request information from SQL about the type and size of each column of the result table.

   Notes:

   a. You can also code the PREPARE statement with an INTO clause to perform the functions of PREPARE and DESCRIBE with a single statement.

   b. If the SQLDA is not large enough to contain column descriptions for each retrieved column, the program must determine how much space is needed, get storage for that amount of space, build a new SQLDA, and reissue the DESCRIBE statement.

5. Allocate the amount of storage needed to contain a row of retrieved data.

6. Put storage addresses into the SQLDA (SQL descriptor area) to tell SQL where to put each item of retrieved data.
7. Open the cursor (declared in step 3) that includes the name of the dynamic select-statement.
8. FETCH a row.
9. When end of data occurs, close the cursor.
10. Handle any SQL return codes that might result.

The SQL Descriptor Area (SQLDA)

You can use the SQLDA to pass information about an SQL statement between SQL and your application.

The SQLDA is a collection of variables required for running the SQL DESCRIBE statement, and may optionally be used by the PREPARE, OPEN, FETCH, and EXECUTE statements. An SQLDA communicates with dynamic SQL. It can be used in a DESCRIBE statement, changed with the addresses of host variables, and then reused in a FETCH statement.

The meaning of the information in an SQLDA depends on its use. In PREPARE and DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In OPEN, EXECUTE, and FETCH, an SQLDA provides information about host variables.

If your application lets you have several cursors open at the same time, you can code several SQLDAs, one for each dynamic select-statement. For more information on SQLDA and SQLCA, see the *SQL/400 Reference* manual.

SQLDAs can be used in RPG and COBOL applications as well as PL/I and C; however, because RPG and COBOL provide no way to set pointers, pointers must be set outside the RPG or COBOL program by a PL/I or C program. Since the area used for the SQLDA must be declared by the PL/I or C program (to get the space for pointers on a 16-byte boundary), the PL/I or C program must do the call of the RPG or COBOL program.

SQLDA Format

The SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of six variables collectively named SQLVAR. When an SQLDA is used in OPEN, FETCH, and EXECUTE, each occurrence of SQLVAR describes a host variable.

The variables of SQLDA are as follows (variable names are in lowercase for C):

SQLDAID
: SQLDAID is used for storage dumps. It is a string of 8 characters that have the value 'SQLDA' after the SQLDA that is used in a PREPARE or DESCRIBE statement. It is not used for FETCH, OPEN, or EXECUTE.

SQLDABC
: SQLABC indicates the length of the SQLDA. It is a 4-byte integer that has the value SQLN*LENGTH(SQLVAR) + 16 after the SQLDA is used in a PREPARE or DESCRIBE statement. SQLDABC must have a value equal to or greater than SQLN*LENGTH(SQLVAR) + 16 prior to use by FETCH, OPEN, or EXECUTE.

SQLN — SQLN is a 2-byte integer that specifies the total number of occurrences of SQLVAR. It must be set prior to use by any SQL statement to a value greater than or equal to 0.

SQLD — SQLD is a 2-byte integer that specifies the pertinent number of occurrences of SQLVAR; that is, the number of host variables described by the SQLDA. This field is set by SQL on a DESCRIBE or PREPARE statement. In other statements, this field must be set prior to use to a value greater than or equal to 0 and less than or equal to SQLN.

SQLVAR — The variables of SQLVAR are SQLTYPE, SQLLEN, SQLRES, SQLDATA, SQLIND, and SQLNAME. These variables are set by SQL on a DESCRIBE or PREPARE statement. In other statements, they must be set prior to use. These variables are defined as follows:

SQLTYPE — SQLTYPE is a 2-byte integer that specifies the data type of the host variable as shown in Table 9-2:

Table 9-2. SQLTYPE Values

Values	Data Type	Null Indicator? (for each value)
500/501	2-byte integer	No/Yes
496/497	4-byte integer	No/Yes
484/485	Decimal	No/Yes
488/489	Numeric (zoned)	No/Yes
480/481	4-byte and 8-byte floating point (IEEE)	No/Yes
452/453	Fixed-length character	No/Yes
448/449	Varying-length character	No/Yes
456/457	Long varying-length character (more than 254 bytes)	No/Yes
460/461	Varying-length character null-terminated character, C only	No/Yes

SQLLEN — SQLLEN is a 2-byte integer variable that is always pertinent, based on the following conditions:

- If the data type is decimal or numeric, the first byte is the precision and the second byte is the scale.
- If the data type is fixed-length character or floating point, SQLLEN is the length of the host variable or column.
- If the data type is varying-length character, SQLLEN must be from 1 to 254 bytes, which indicates the maximum length of the host variable.
- If the data type is long varying-length character, SQLLEN must be from 1 to 32 766 bytes, which indicates the maximum length of the host variable.
- If the data type is a 2-byte or 4-byte integer, SQLLEN is either the length of the host variable or column; or precision and scale. If the first byte is 0, SQLLEN contains a length of 2 or

4. If the first byte is nonzero, the first byte is the precision and the second byte is the scale.

SQLRES
: SQLRES is a 12-byte reserved area for boundary alignment purposes. Note that pointers *must* be on a quad-word boundary.

SQLDATA
: SQLDATA is a 16-byte pointer variable that specifies the address of the host variables when the SQLDA is used on OPEN, FETCH, and EXECUTE.

When the SQLDA is used on PREPARE and DESCRIBE, this area is overlaid with the following information:

- The third and fourth bytes contain a small integer and indicate whether the column is FOR BIT DATA. If the small integer is -1, the column is bit data (FOR BIT DATA).

SQLIND
: SQLIND is a 16-byte pointer that specifies the address of a small integer host variable that is used as an indication of null or not null when the SQLDA is used on OPEN, FETCH, and EXECUTE. A negative value indicates null and a nonnegative indicates not null. Since the SQL/400 program does not provide full null value support, this has limited usefulness. There are cases, however, when the result of a query is a null value even though nulls cannot be stored in tables. These cases are:

- If a group function (MIN, MAX, and so on) is specified in the SELECT list, and the GROUP BY clause is not specified, and the result of COUNT is 0, then a null value (−1) is returned in the indicator variable for the group functions other than COUNT.
- If a decimal data error occurred when evaluating an expression in the SELECT list, but a successful analysis could still be made as to whether the resulting row should be selected, as many valid results are returned as possible, and items that encountered errors are returned as a null value (−2).

When the SQLDA is used on PREPARE and DESCRIBE, this area is reserved for future use.

SQLNAME
: SQLNAME is a variable-length character variable with a maximum length of 30, which contains the name of selected column or label names after a PREPARE or DESCRIBE. It is reserved in OPEN, FETCH, or EXECUTE.

Example of a Select-Statement for Allocating Storage for SQLDA

The select-statement can be read from a display station or from a host variable, or it can be developed within an application program. The following example shows a select-statement read from a display station:

```
SELECT DEPTNO, PHONENO FROM USER1.TEMPL
  WHERE LASTNAME = 'PARKER'
```

Note: The select-statement has no INTO clause. Dynamic select-statements must *not* have an INTO clause, even if they return only one row.

When the statement is read, it is assigned to a host variable. The host variable (for example, named DSTRING) is then processed, using the PREPARE statement, as shown:

```
EXEC SQL
  PREPARE S1 FROM :DSTRING;
```

Allocating Storage

Now you can allocate storage for the SQLDA. The techniques for acquiring storage are language dependent. The SQLDA must be allocated on a 16-byte boundary. The SQLDA consists of a fixed-length header, 16 bytes long. The header is followed by a varying-length array section (SQLVAR), each element of which is 80 bytes in length. The amount of storage you need to allocate depends on how many elements you want to have in the SQLVAR array. Each column you select must have a corresponding SQLVAR array element. Therefore, the number of columns listed in your select-statement determines how many SQLVAR array elements you should allocate. Because select-statements are specified at run time, however, it is impossible to know how many columns will be accessed. Consequently, you must estimate the number of columns. Suppose, in this example, that no more than 20 columns are ever expected to be accessed by a single select-statement. This means that the SQLVAR array should have a dimension of 20 (for an SQLDA size 20 x 80, or 1600, plus 16 for a total of 1616 bytes), because each item in the select-list must have a corresponding entry in SQLVAR.

Having allocated what you estimated to be enough space for your SQLDA, in the SQLN field of the SQLDA, set an initial value equal to the number of SQLVAR array elements. In the following example, set SQLN to 20:

```
Allocate space for an SQLDA of 1616 bytes on a quadword boundary
SQLN = 20;
```

Having allocated storage, you can now issue a DESCRIBE statement.

```
EXEC SQL
  DESCRIBE S1 INTO :SQLDA;
```

When the DESCRIBE statement is run, SQL places values in the SQLDA that provide information about the select-list. The following Figure 9-1 shows the contents of the SQLDA after the DESCRIBE is run:

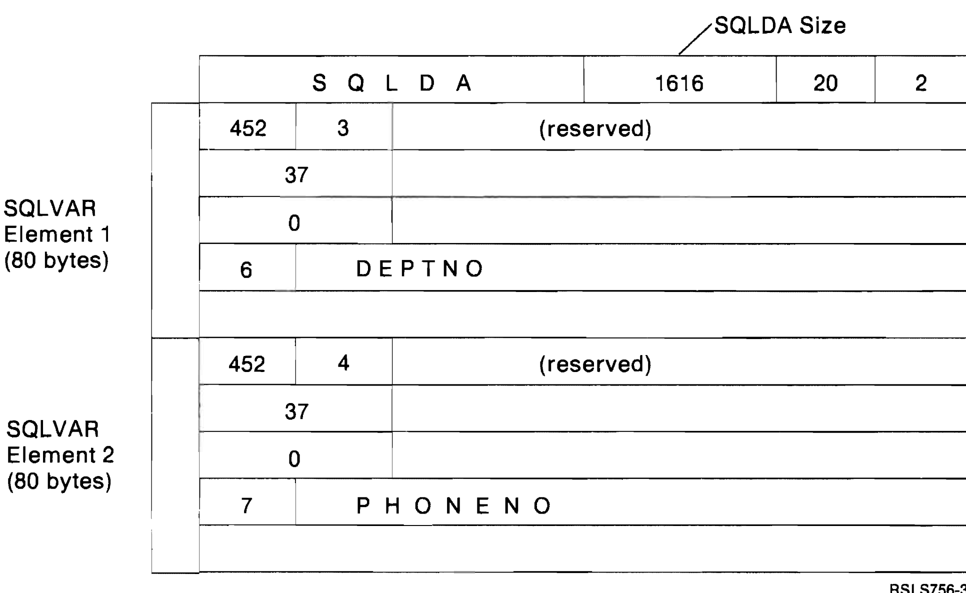


Figure 9-1. Contents of SQLDA after a DESCRIBE Is Run

SQLDAID is an identifier field initialized by SQL when a DESCRIBE is run. SQLDABC is the byte count or size of the SQLDA. You can ignore these for now.

The example for running the SELECT statement for S1 is:

```
SELECT DEPTNO, PHONENO
  FROM USER1.TEMPL
  WHERE LASTNAME = 'PARKER'
```

Your program might have to alter the SQLN value if the SQLDA is not large enough to contain the described SQLVAR elements. For example, let the select-statement contain 27 select-list expressions instead of the 20 or less that you estimated. Because the SQLDA was only allocated with an SQLVAR dimension of 20 elements, SQL cannot describe the select-list, because the SQLVAR has too many elements. SQL sets the SQLD to the actual number of columns specified by the SELECT statement, and the rest of the structure is ignored. Therefore, after a DESCRIBE, you should compare the SQLN to the SQLD. If the value of SQLD is greater than the value of SQLN, allocate a larger SQLDA based on the value in SQLD, as follows:

```
EXEC SQL
  DESCRIBE S1 INTO :SQLDA;
IF SQLN <= SQLD THEN
DO;

  /*Allocate a larger SQLDA using the value of SQLD.*/
  /*Reset SQLN to the larger value.*/

EXEC SQL
  DESCRIBE S1 INTO :SQLDA;
END;
```

If you use DESCRIBE on a non-SELECT statement, SQL sets SQLD to 0. Therefore, if your program is designed to process both SELECT and nonSELECT statements, you can describe each statement (after it is prepared) to determine whether it is a select-statement. This sample routine is designed to process only select-statements; the SQLD is not checked.

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 DEPTNO, PHONENO
  FROM USER1.TEMPL
  WHERE LASTNAME = 'PARKER'
```

The first item in the select-list is DEPTNO. At the beginning of this section, we identified that each SQLVAR element contains the fields SQLTYPE, SQLLEN, SQLRES, SQLDATA, SQLIND, and SQLNAME. SQL returns, in the SQLTYPE field, a code that describes the data type of the expressions and whether nulls are applicable or not.

For example, SQL sets SQLTYPE to 452 in SQLVAR element 1 (see Figure 9-1 on page 9-11). This specifies that DEPTNO is a fixed-length character string (CHAR) column and that nulls are not permitted in the column.

SQL sets SQLLEN to the length of the column. For character strings, SQLLEN is set to the maximum length of the character string. For decimal, numeric, or nonzero scale binary, the precision and scale are returned in the first and second bytes, respectively. For other data types, SQLLEN is set as follows:

```
SMALLINT -- SQLLEN = 2
INTEGER  -- SQLLEN = 4
REAL     -- SQLLEN = 4
FLOAT    -- SQLLEN = 8
```

Because the data type of DEPTNO is CHAR, SQL sets SQLLEN equal to the length of the character string. For DEPTNO, that length is 3. Therefore, when the SELECT statement is later run, a storage area large enough to hold a CHAR(3) string is needed.

Because the data type of DEPTNO is CHAR FOR SBCS DATA, the first 4 bytes of SQL data were set to a nonzero value (see Figure 9-1 on page 9-11). The last field in an SQLVAR element is a varying-length character string called SQLNAME. The first 2 bytes of SQLNAME contain the length of the character data. The character data itself is usually the name of a column used in the select statement (DEPTNO in the above example). The exceptions to this are select-list items that are unnamed, such as functions (for example, SUM(SALARY)), expressions (for example, A+B−C), and constants. In these cases, SQLNAME is an empty string. SQLNAME may also contain a label rather than a name. One of the parameters associated with the PREPARE and DESCRIBE statements is the USING clause. You can specify it this way:

```
EXEC SQL
  DESCRIBE S1 INTO SQLDA
    USING LABELS;
```

If you specify NAMES (or omit the USING parameter entirely), only column names are placed in the SQLNAME field. If you specify LABELS, only labels associated with the columns listed in your SQL statement are entered here. If you specify ANY, labels are placed in the SQLNAME field for those columns that have labels; other-

wise, the column names are entered. If you specify BOTH, names and labels are both placed in the field with their corresponding lengths. If you specify BOTH, however, you must remember to double the size of the SQLVAR array because you are including twice the number of elements. For more information on the USING option and on column labels, see the *SQL/400 Reference* manual.

In the example, the second SQLVAR element contains the information for the second column used in the select: PHONENO. The 452 code in SQLTYPE specifies that PHONENO is a CHAR column. For a CHAR data type of length 4, SQL sets SQLLEN to 4.

After analyzing the result of the DESCRIBE, you can allocate storage for variables containing the result of the SELECT statement. For DEPTNO, a character field of length 3 must be allocated; for PHONENO, a character field of length 4 must be allocated.

After 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 place where the results are to be put. SQLIND points to the place where the null indicator is to be put. The following figure shows what the structure looks like now:

SQLDA Size

	S Q L D A	1616	20	2
SQLVAR Element 1 (80 bytes)	452	3	(reserved)	
	Address of FLDA			
	Address of FLDAI			
	6	D E P T N O		
SQLVAR Element 2 (80 bytes)	452	4	(reserved)	
	Address of FLDB			
	Address of FLDBI			
	7	P H O N E N O		

FLDA: (CHAR(3))

FLDB: (CHAR(4))

Indicator Variables: (halfword)
FLDAI: FLDBI:

RSLS757-2

This is what was done so far:

```
EXEC SQL
  INCLUDE SQLDA;
/*Read a statement into the DSTRING varying-length
  character string host variable.*/
EXEC SQL
  PREPARE S1 FROM :DSTRING;
/*Allocate an SQLDA of 1616 bytes.*/
SQLN =20;
EXEC SQL
  DESCRIBE S1 INTO :SQLDA;
/*Analyze the results of the DESCRIBE.*/
/*Allocate storage to hold one row of
  the result table.*/
/*Set SQLDATA and SQLIND for each column
  of the result table.*/
```

Using a Cursor

You are now ready to retrieve the select-statements results. Dynamically defined select-statements must not have an INTO statement. Therefore, all dynamically defined select-statements must use a cursor. Special forms of the DECLARE, OPEN, and FETCH are used for dynamically defined select-statements.

The DECLARE statement for the example statement is:

```
EXEC SQL 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 actual retrieval of result rows is made as follows:

```
EXEC SQL
  OPEN C1;
EXEC SQL
  FETCH C1 USING DESCRIPTOR :SQLDA;
DO WHILE (SQLCODE = 0);
  /*Display ... the results pointed to by SQLDATA*/
  END;
  /*Display ('END OF LIST')*/
EXEC SQL
  CLOSE C1;
```

The cursor is opened, and the result table is evaluated. Notice that there are no input host variables needed for the example select-statement. The SELECT result rows are then returned using FETCH. On the FETCH statement, there is no list of output host variables. Rather, the FETCH statement tells SQL to return results into areas described by 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 following figure shows what the structure looks like after the FETCH statement has been processed.

SQLDA Size

S Q L D A	1616	20	2

SQLVAR Element 1 (80 bytes)

452	3	(reserved)
Address of FLDA		
Address of FLDAI		
6	DEPTNO	

SQLVAR Element 2 (80 bytes)

452	4	(reserved)
Address of FLDB		
Address of FLDBI		
7	PHONENO	

FLDA: (CHAR(3)) E11

FLDB: (CHAR(4)) 4502

Indicator Variables: (halfword)

FLDAI: 0 FLDBI: 0

RSLS758-2

The meaning of the SMALLINT 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 furnished was not large enough.
    The indicator variable contains the length before
    truncation.
```

Note: Unless HOLD is specified, dynamic cursors are closed and prepared statements are destroyed during COMMIT or ROLLBACK. Before opening the cursor, you must issue a PREPARE to use the statement again, if HOLD was not specified on COMMIT or ROLLBACK.

Using Parameter Markers

In the example we are using, the select-statement that was dynamically run had predictable parameters (input host variables) in the WHERE clause. In the example, it was:

```
WHERE LASTNAME = 'PARKER'
```

If you want to run the same select-statement several times, using different values for LASTNAME, you can use an SQL statement such as PREPARE or EXECUTE (as described in "Using the PREPARE and EXECUTE Statements" on page 9-4) like this:

```
SELECT DEPTNO, PHONENO FROM USER1.TEMPL WHERE LASTNAME = ?
```

When your parameters are not predictable, your application cannot know the number or types of the parameters until run time. You can arrange to receive this information at the time your application is run, and by using a USING DESCRIPTOR on the OPEN CURSOR statement, you can substitute the values contained in specific host variables for the parameter markers included in the WHERE clause of the select-statement.

To code such a program, you need to use the OPEN CURSOR statement with the USING DESCRIPTOR clause. This SQL statement is used to not only open a cursor, but to replace each parameter marker with the value of the corresponding host vari-

able. The descriptor name that you specify with this statement must identify an SQLDA that contains a valid description of those host variables. This SQLDA, unlike those previously described, is not used to return information on data items that are part of a SELECT list. That is, it is not used as output from a DESCRIBE statement, but as input to the OPEN CURSOR statement. It provides information on host variables that are used to replace parameter markers in the WHERE clause of the SELECT statement. It gets this information from the application, which must be designed to place appropriate values into the necessary fields of the SQLDA. The SQLDA is then ready to be used as a source of information for SQL in the process of replacing parameter markers with host variable data.

When you use the SQLDA for input to the OPEN CURSOR statement with the USING DESCRIPTOR clause, not all of its fields have to be filled in. Specifically, SQLDAID, SQLRES, and SQLNAME can be left blank. Therefore, when you use this method for replacing parameter markers with host variable values, you need to determine:

- How many ? parameter markers are there?
- What are the data types and attributes of these parameters markers (SQLTYPE and SQLLEN)?
- Do you want an indicator variable?

In addition, if the routine is to handle both SELECT and nonSELECT statements, you may want to determine what category of statement it is. (Alternatively, you can write code to look for the SELECT keyword.)

If your application uses parameter markers, your program has to:

1. Read a statement into the DSTRING varying-length character string host variable.
2. Determine the number of ? parameter markers.
3. Allocate an SQLDA of that size.
4. Set SQLN and SQLD to the number of ? parameter markers.
5. Set SQLDABC equal to SQLN*LENGTH(SQLVAR) + 16.
6. For each ? parameter marker:
   a. Determine the data types, lengths, and indicators.
   b. Set SQLTYPE and SQLLEN.
   c. Allocate storage to hold the input values (the ? values).
   d. Set these values.
   e. Set SQLDATA and SQLIND (if applicable) for each ? parameter marker.
   f. Issue the OPEN CURSOR statement with a USING DESCRIPTOR clause to open your cursor and substitute a host variable value for each of the parameter markers.

The statement can then be processed normally.

Chapter 10. Preparing and Running a Program with SQL Statements

This chapter describes some of the tasks for preparing and running an application program. The tasks described are:

- Precompiling
- Compiling
- Binding
- Running

Basic Processes of the SQL Precompiler

You must precompile and compile an application program containing embedded SQL statements before you can run it. Precompiling of such programs is done by the SQL precompiler. The SQL precompiler scans each statement of the application program source and does the following:

- **Looks for SQL statements and for the definition of host variable names.** The variable names and definitions are used to verify the SQL statements. You can examine the listing after the SQL precompiler completes processing to see if any errors occurred.

- **Verifies that each SQL statement is valid and free of syntax errors.** The validation procedure supplies error messages in the output listing that helps you correct any errors that occur.

- **Validates the SQL statements using the description in the database.** During the precompile, the SQL statements are checked for valid table, view, and column names. If a referred to table or view does not exist, or you are not authorized to the table or view at the time of the precompile or compile, the validation is done at run time. If the table or view does not exist at run time, an error occurs.

  Notes:

  1. Overrides are processed when retrieving external definitions. For more information, see the *Database Guide* and the *Data Management Guide.*

  2. You need some authority (at least *OBJOPR) to any tables or views referred to in the SQL statements in order to validate the SQL statements. The actual authority required to process any SQL statement is checked at run time. For more information on any SQL statement, see the *SQL/400 Reference* manual.

- **Prepares each SQL statement for compilation in the host language.** For most SQL statements, the SQL precompiler inserts a comment and a CALL statement to the SQL interface module (QSQROUTE). For some SQL statements (for example, DECLARE statements), the SQL precompiler produces no host language statement except a comment.

- **Produces information about each precompiled SQL statement.** The information is stored internally in a temporary source file member, where it is available for use during the bind process.

To get complete diagnostic information when you precompile, specify the *SOURCE and *XREF precompiler options.

© Copyright IBM Corp. 1988, 1989

Input to the Precompiler

Application programming statements and embedded SQL statements are the primary input to the SQL precompiler. In PL/I and C programs, the SQL statements must use the same margins as specified in the MARGINS parameter of the CRTSQLPLI and CRTSQLC commands.

The SQL precompiler assumes that the host language statements are syntactically correct. If the host language statements are not syntactically correct, the precompiler may not correctly identify SQL statements and host variable declarations. There are limits on the forms of source statements that can be passed through the precompiler. Literals and comments, that are not accepted by the application language compiler, can interfere with the precompiler source scanning process and cause errors.

The SQL INCLUDE statement can be used to get secondary input from the file specified by the INCFILE parameter of the CRTSQLxxx[1] command. The SQL INCLUDE statement causes input to be read from the specified member until the end of the member is reached. The included member may not contain other precompiler INCLUDE statements, but can contain both application program and SQL statements.

Another preprocessor may process source statements before the SQL precompiler. However, any preprocessor run before the SQL precompile must be able to pass through SQL statements.

If double-byte character set (DBCS) literals are specified in the application program source, the system value QIGC (for using DBCS characters) must indicate that the system supports DBCS literals.

Output from the Precompiler

The following sections describe the various kinds of output supplied by the precompiler.

Listing

The output listing is sent to the print file specified by the PRTFILE parameter of the CRTSQLxxx command. The following items are output to the printer file:

- Precompiler options

  Options specified in the CRTSQLxxx command.

- Precompiler source

  This output supplies precompiler source statements, with record numbers assigned by the precompiler, if the *SOURCE option is in effect.

- Precompiler cross-reference

  This output supplies a cross-reference listing (if *XREF was specified in the OPTION parameter), showing the precompiler line numbers of SQL statements in which host names and column names are referred to.

[1] The xxx in this command refers to the host language indicators: CBL for COBOL/400, PLI for AS/400 PL/I, C for C/400 and RPG for RPG/400.

- Precompiler diagnostics

  This output supplies diagnostic messages, showing the precompiler record numbers of statements in error.

Temporary Source File Members

Source statements processed by the precompiler are written to QSQLTEMP in the QTEMP library. In your precompiler-changed source code, SQL statements have been converted to comments and calls to the SQL interface module, QSQROUTE, application language interface. The name of the temporary source file member is the same as the name specified in the PGM parameter of the CRTSQLxxx. This member cannot be changed before being used as input to the compiler.

QSQLTEMP can be moved to a permanent library after the precompile, if you want to compile at a later time. You cannot change the records of the source member, or the attempted compile will fail.

Sample Precompiler Output

The output can provide information about your precompiled source module if you specify the *SOURCE (*SRC) and *XREF options on the OPTION parameter when you call the SQL precompiler.

The format of the precompiler output is:

```
     5728ST1 R02 M00 891006                        IBM SQL/400       CBLTEST1                  89-03-28 15:44:34   Page   1
    ┌Source type..............COBOL
    │Program name.............USER1/CBLTEST1
    │Source file..............*LIBL/QLBLSRC
    │Member...................*PGM
    │Options..................*SRC      *XREF     *SQL
 1 <│Target release...........*CURRENT
    │INCLUDE file.............*LIBL/*SRCFILE
    │Commit...................*CHG
    │Generation level.........10
    │Printer file.............*LIBL/QSYSPRT
    └Text.....................*SRCMBRTXT
     Source member changed on 89-03-24  13:06:14
                               2
```

RSLS760-3

1 A list of the options you specified when the SQL precompiler was called.

2 The date the source member was last changed.

```
5728ST1 R02 M00 891006                    IBM SQL/400       CBLTEST1                    89-03-28 15:44:34   Page    2
  1                                                                                2        3
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
     1         IDENTIFICATION DIVISION.                                                      100   03-24-89
     2         PROGRAM-ID.  CBLTEST1.                                                        200   03-24-89
     3         ENVIRONMENT DIVISION.                                                         300   03-24-89
     4         CONFIGURATION SECTION.                                                        400   03-24-89
     5         SOURCE-COMPUTER. IBM-AS400.                                                   500   03-24-89
     6         OBJECT-COMPUTER. IBM-AS400.                                                   600   03-24-89
     7         INPUT-OUTPUT SECTION.                                                         700   03-24-89
     8         FILE-CONTROL.                                                                 800   03-24-89
     9             SELECT OUTFILE, ASSIGN TO PRINTER-QPRINT,                                 900   03-24-89
    10                FILE STATUS IS FSTAT.                                                 1000   03-24-89
    11         DATA DIVISION.                                                               1100   03-24-89
    12         FILE SECTION.                                                                1200   03-24-89
    13         FD  OUTFILE                                                                  1300   03-24-89
    14             DATA RECORD IS REC-1,                                                    1400   03-24-89
    15             LABEL RECORDS ARE OMITTED.                                               1500   03-24-89
    16         01  REC-1.                                                                   1600   03-24-89
    17             05  CC                     PIC X.                                        1700   03-24-89
    18             05  DEPT-NO                PIC X(3).                                     1800   03-24-89
    19             05  FILLER                 PIC X(5).                                     1900   03-24-89
    20             05  AVERAGE-EDUCATION-LEVEL PIC ZZZ.                                     2000   03-24-89
    21             05  FILLER                  PIC X(5).                                    2100   03-24-89
    22             05  AVERAGE-SALARY          PIC ZZZZ9.99.                                2200   03-24-89
    23         01  ERROR-RECORD.                                                            2300   03-24-89
    24             05  CC                      PIC X.                                       2400   03-24-89
    25             05  ERROR-CODE              PIC S9(5).                                   2500   03-24-89
    26             05  ERROR-MESSAGE           PIC X(70).                                   2600   03-24-89
    27         WORKING-STORAGE SECTION.                                                     2700   03-24-89
    28             EXEC SQL                                                                 2800   03-24-89
    29               INCLUDE SQLCA                                                          2900   03-24-89
    30             END-EXEC.                                                                3000   03-24-89
    31         77  FSTAT                       PIC XX.                                      3100   03-24-89
    32         01  AVG-RECORD.                                                              3200   03-24-89
    33             05  DEPTNO                  PIC X(3).                                    3300   03-24-89
    34             05  AVG-EDUC                PIC S9(4) USAGE COMP-4.                      3400   03-24-89
    35             05  AVG-SALARY              PIC S9(6)V99 COMP-3.                         3500   03-24-89
    36         PROCEDURE DIVISION.                                                          3600   03-24-89
    37        ****************************************************************              3700   03-24-89
    38        *  This program will get the average education level and the  *              3800   03-24-89
    39        *  average salary by department.                              *              3900   03-24-89
    40        ****************************************************************              4000   03-24-89
    41         A000-MAIN-PROCEDURE.                                                         4100   03-24-89
    42             OPEN OUTPUT OUTFILE.                                                     4200   03-24-89
    43        ****************************************************************              4300   03-24-89
    44        * Set-up WHENEVER statement to handle SQL errors.              *              4400   03-24-89
    45        ****************************************************************              4500   03-24-89
    46             EXEC SQL                                                                 4600   03-24-89
    47               WHENEVER SQLERROR GO TO B000-SQL-ERROR                                 4700   03-24-89
    48             END-EXEC.                                                                4800   03-24-89
    49        ****************************************************************              4900   03-24-89
    50        * Declare cursor                                              *              5000   03-24-89
    51        ****************************************************************              5100   03-24-89
    52             EXEC SQL                                                                 5200   03-24-89
    53               DECLARE CURS CURSOR FOR                                                5300   03-24-89
    54                 SELECT DEPTNO, AVG(EDUCLVL), AVG(SALARY)                             5400   03-24-89
    55                   FROM USER1.TEMPL                                                   5500   03-24-89
    56                   GROUP BY DEPTNO                                                    5600   03-24-89
```

```
5728ST1 R02 M00 891006                IBM SQL/400      CBLTEST1                       89-03-28 15:44:34   Page    3
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
   57            END-EXEC.                                                                 5700    03-24-89
   58       ****************************************************************               5800    03-24-89
   59       *  Open cursor                                                  *               5900    03-24-89
   60       ****************************************************************               6000    03-24-89
   61            EXEC SQL                                                                  6100    03-24-89
   62              OPEN CURS                                                               6200    03-24-89
   63            END-EXEC.                                                                 6300    03-24-89
   64       ****************************************************************               6400    03-24-89
   65       *  Fetch all result rows                                        *               6500    03-24-89
   66       ****************************************************************               6600    03-24-89
   67            PERFORM A010-FETCH-PROCEDURE THROUGH A010-FETCH-EXIT                      6700    03-24-89
   68              UNTIL SQLCODE IS = 100.                                                 6800    03-24-89
   69       ****************************************************************               6900    03-24-89
   70       *  Close cursor                                                 *               7000    03-24-89
   71       ****************************************************************               7100    03-24-89
   72            EXEC SQL                                                                  7200    03-24-89
   73              CLOSE CURS                                                              7300    03-24-89
   74            END-EXEC.                                                                 7400    03-24-89
   75            CLOSE OUTFILE.                                                            7500    03-24-89
   76            STOP RUN.                                                                 7600    03-24-89
   77       ****************************************************************               7700    03-24-89
   78       *  Fetch a row and move the information to the output record.  *               7800    03-24-89
   79       ****************************************************************               7900    03-24-89
   80        A010-FETCH-PROCEDURE.                                                         8000    03-24-89
   81            MOVE SPACES TO REC-1.                                                     8100    03-24-89
   82            EXEC SQL                                                                  8200    03-24-89
   83              FETCH CURS INTO :AVG-RECORD                                             8300    03-24-89
   84            END-EXEC.                                                                 8400    03-24-89
   85            IF SQLCODE IS = 0                                                         8500    03-24-89
   86              MOVE DEPTNO TO DEPT-NO                                                  8600    03-24-89
   87              MOVE AVG-SALARY TO AVERAGE-SALARY                                       8700    03-24-89
   88              MOVE AVG-EDUC TO AVERAGE-EDUCATION-LEVEL                                8800    03-24-89
   89              WRITE REC-1 AFTER ADVANCING 1 LINE.                                     8900    03-24-89
   90        A010-FETCH-EXIT.                                                              9000    03-24-89
   91            EXIT.                                                                     9100    03-24-89
   92       ****************************************************************               9200    03-24-89
   93       *  An SQL error occurred.  Move the error number to the error  *               9300    03-24-89
   94       *  record and stop running.                                     *               9400    03-24-89
   95       ****************************************************************               9500    03-24-89
   96        B000-SQL-ERROR.                                                               9600    03-24-89
   97            MOVE SPACES TO ERROR-RECORD.                                              9700    03-24-89
   98            MOVE SQLCODE TO ERROR-CODE.                                               9800    03-24-89
   99            MOVE "AN SQL ERROR HAS OCCURRED" TO ERROR-MESSAGE.                        9900    03-24-89
  100            WRITE ERROR-RECORD AFTER ADVANCING 1 LINE.                               10000    03-24-89
  101            CLOSE OUTFILE.                                                           10100    03-24-89
  102            STOP RUN.                                                                10200    03-24-89
                          * * * * *  E N D  O F  S O U R C E  * * * * *
```

1 The record number assigned by the precompiler when it reads the source record. The record numbers are used to identify the source record in error messages and SQL runtime processing.

2 The sequence number taken from the source record. This is the number seen when you use the source entry utility (SEU) to edit the source member.

3 The date when the source record was last changed. If the record is blank, it indicates that the record has not been changed since it was created.

```
5728ST1 R02 M00 891006                          IBM SQL/400       CBLTEST1                              89-03-28 15:44:34   Page      4
                                                CROSS REFERENCE
   1                             2               3
Data Names                     Define         Reference
'CURS'                           53        4  CURSOR
                                           5  62 73 83
'DEPTNO'                          ****        COLUMN
                                              54 56
'EDUCLVL'                         ****        COLUMN
                                              54
'SALARY'                          ****        COLUMN
                                              54
'TEMPL'                           ****        TABLE IN 'USER1'
                                              55
'USER1'                           ****        COLLECTION
                                              55
AVERAGE-EDUCATION-LEVEL           20          IN REC-1
AVERAGE-SALARY                    22          IN REC-1
                                                      6                            7
AVG-EDUC                          34          SMALL INTEGER PRECISION(4,0) IN AVG-RECORD
AVG-RECORD                        32          STRUCTURE
                                              83
AVG-SALARY                        35          DECIMAL(8,2) IN AVG-RECORD
B000-SQL-ERROR                    ****        LABEL
                                              47
CC                                17          CHARACTER(1)  IN REC-1
CC                                24          CHARACTER(1)  IN ERROR-RECORD
DEPT-NO                           18          CHARACTER(3)  IN REC-1
DEPTNO                            33          CHARACTER(3)  IN AVG-RECORD
ERROR-CODE                        25          NUMERIC(5,0)  IN ERROR-RECORD
ERROR-MESSAGE                     26          CHARACTER(70)  IN ERROR-RECORD
ERROR-RECORD                      23          STRUCTURE
FSTAT                             31          CHARACTER(2)
REC-1                             16
```

1 Data names are the symbolic names used in source statements. Names enclosed in quotation marks (") or apostrophes (') are names of SQL entities, such as tables and columns. Names not enclosed by quotation marks or apostrophes are host variables.

2 The define column specifies the line number at which the name is defined. The line number is generated by the SQL precompiler. **** means that the object was not defined or the precompiler did not recognize the declarations.

3 The reference column contains two types of information:

- What the symbolic name is defined as 4
- The line numbers where the symbolic name occurs 5

If the symbolic name refers to a valid host variable, the data type 6 or structure 7 is also noted.

```
5728ST1 R02 M00 891006                         IBM SQL/400       CBLTEST1                          89-03-28 15:44:34   Page     5
                                             DIAGNOSTIC MESSAGES
MSG ID  SEV  RECORD  TEXT
SQL1103  10      55  Position 23 Field definitions for file TEMPL in USER1 not found.
                                               Message Summary
Total    Info    Warning    Error    Severe    Terminal
   1      0         1         0        0          0
  102 Source records processed
                               * * * * *  E N D  O F  L I S T I N G  * * * * *
```

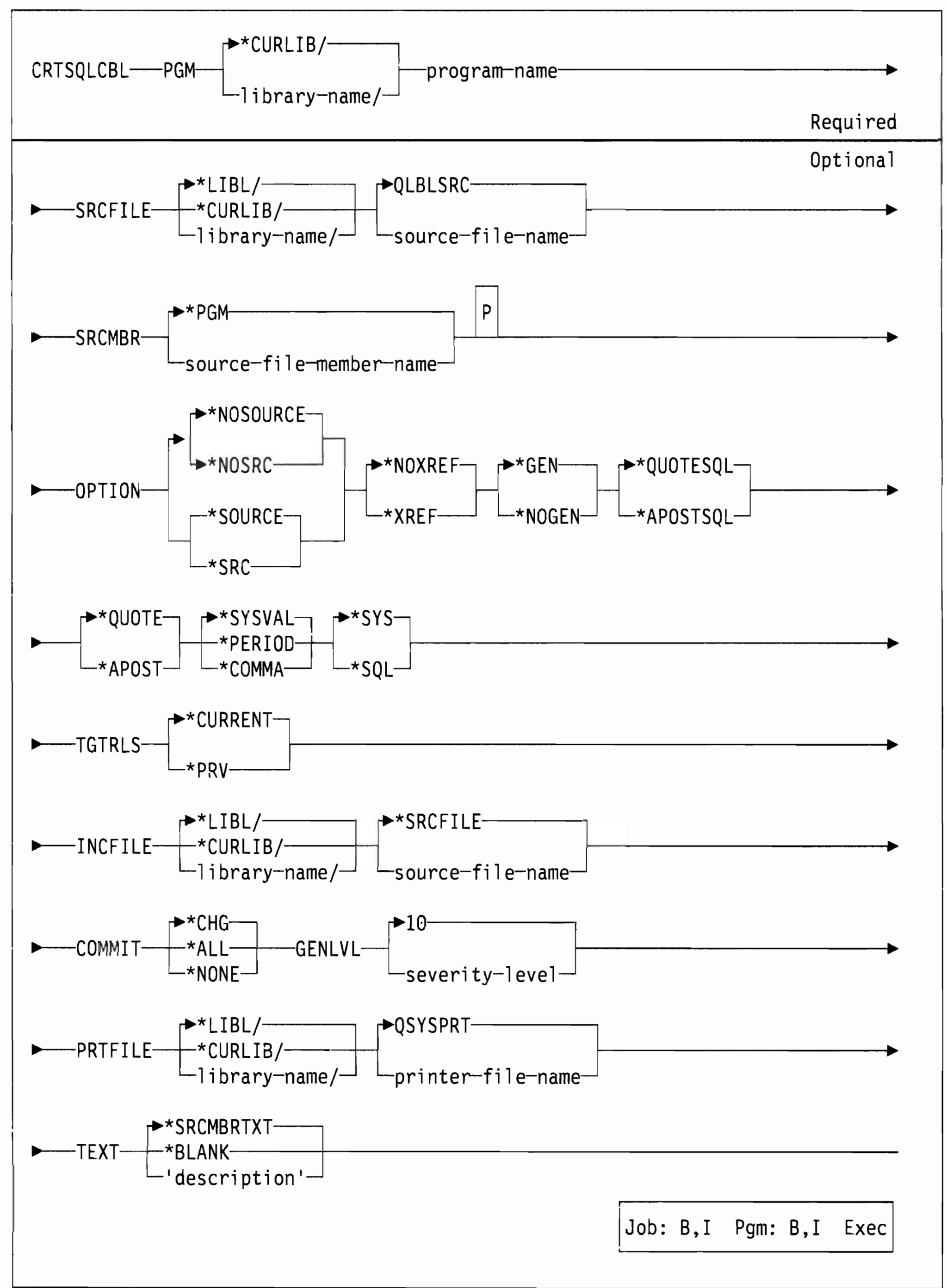
Precompiler Commands

In the SQL/400 program there are four precompiler commands (CRTSQLCBL, CRTSQLPLI, CRTSQLC, and CRTSQLRPG); one for each of the host languages: COBOL/400, AS/400 PL/I, C/400, and RPG III (part of RPG/400). Separate commands by language let you specify the required parameters and then take the default for the remaining parameters, because the defaults are applicable only to the one language you are using. For example, the options *APOST and *QUOTE are unique to COBOL. They are not included in the commands for the other languages.

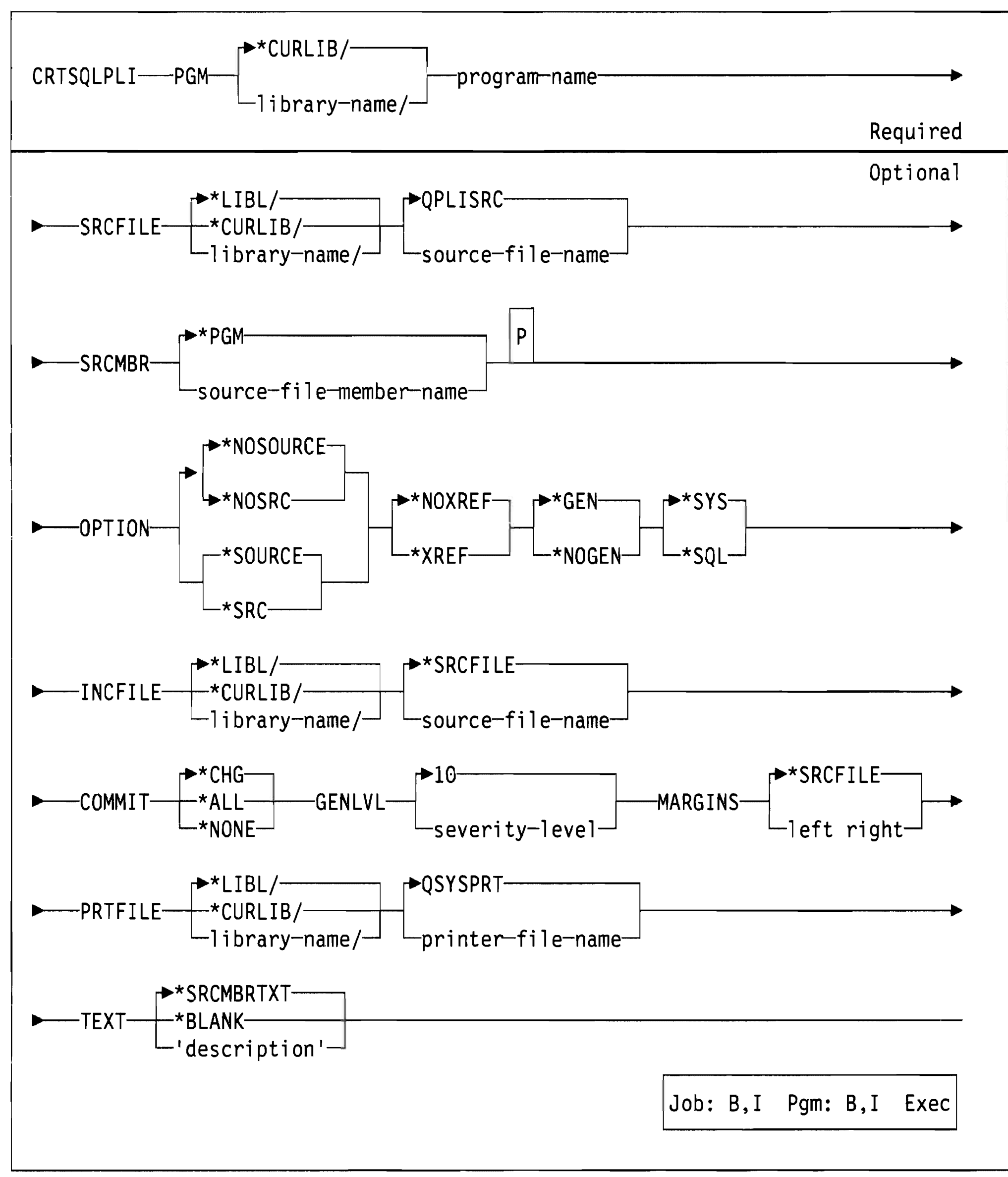
Syntax for the Precompiler Commands

The syntax for the SQL/400 precompiler commands is shown on the following pages. The parameters are defined under "Parameter Definitions" on page 10-12.

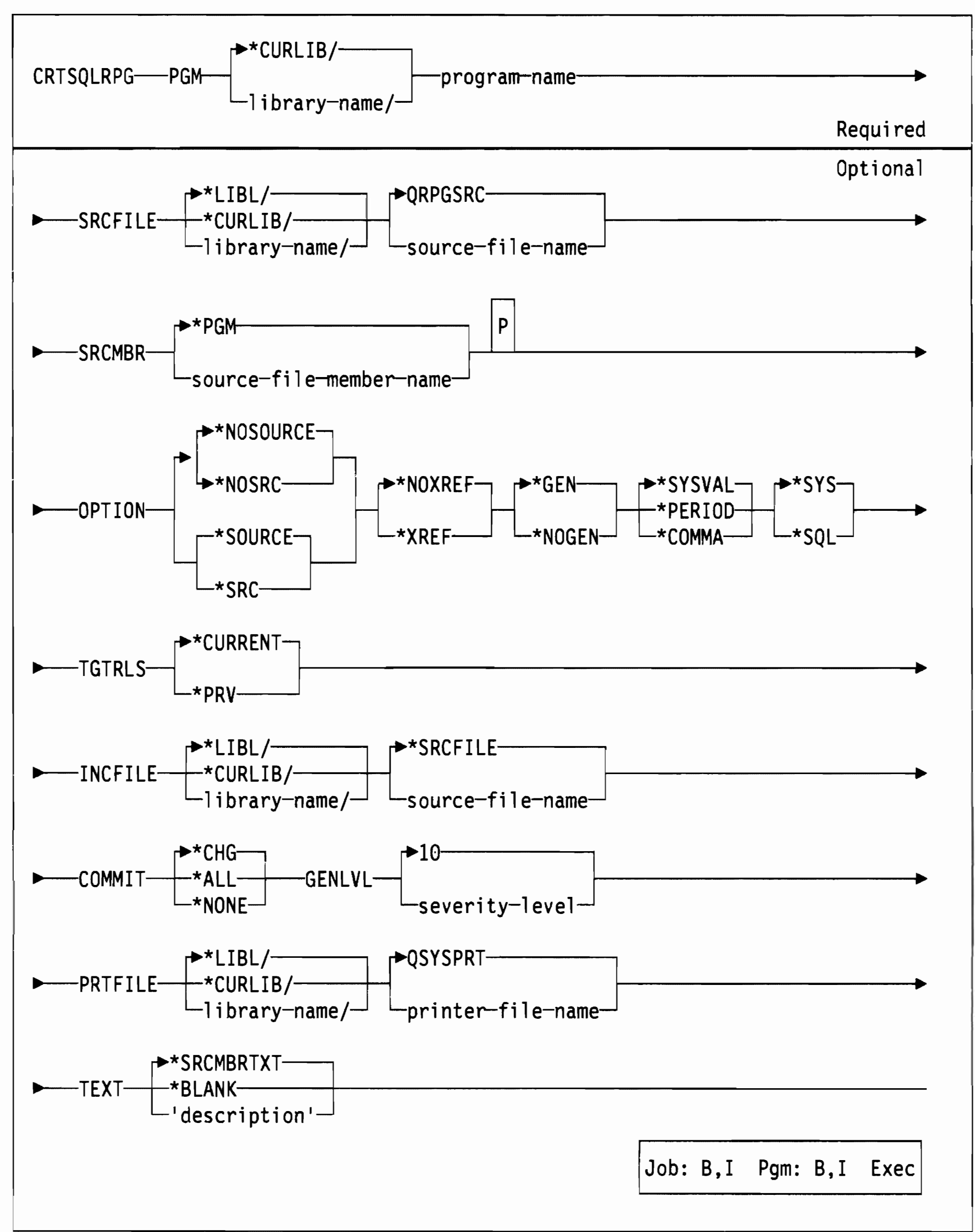
The syntax for the create SQL COBOL (CRTSQLCBL) precompiler command is:



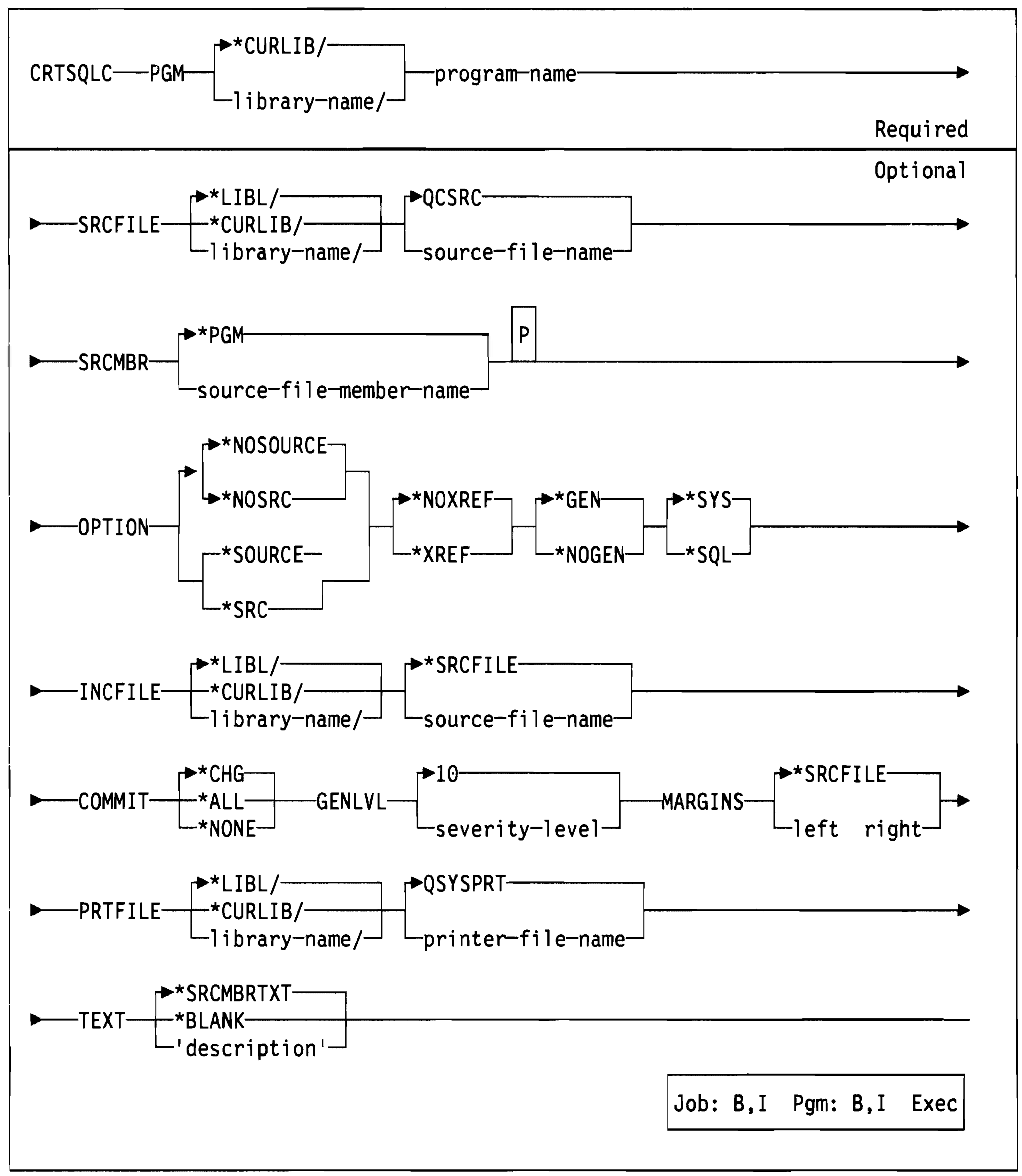
The syntax for the create SQL PL/I (CRTSQLPLI) precompiler command is:



The syntax for the create SQL RPG (CRTSQLRPG) precompiler command is:



The syntax for the create SQL C (CRTSQLC) precompiler command is:



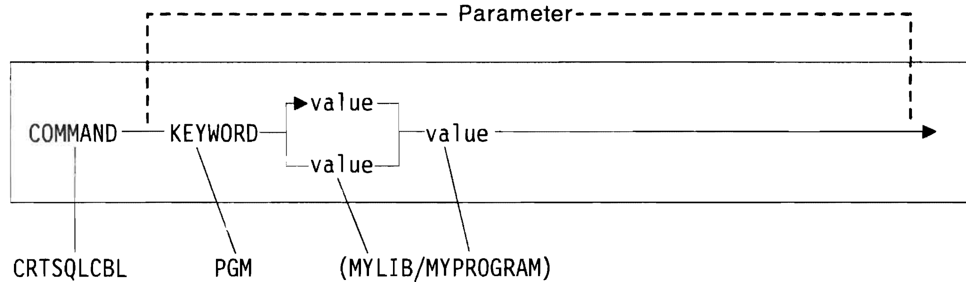
Precompiler Command Parameters

The precompiler is controlled by parameters specified when it is called by one of the SQL precompiler commands. The parameters specify how the input is processed, and how the output is presented.

You can precompile a program without specifying anything more than the name of the member containing the program source statements as the PGM parameter of the CRTSQLxxx. SQL assigns default values for all precompiler parameters (which may, however, be overridden by any that you explicitly specify).

Parameter Definitions

The following paragraphs describe the parameters for the precompiler commands. The parameters consist of **keywords** and **values**. The following figure defines the parts of a parameter as used throughout this manual:



Note: The default value is indicated by an arrowhead (→).

RSLS759-2

Required Parameter

The following parameter is required for the SQL precompiler commands. If you choose not to specify any of the following keyword values, then the defaults for those values are used.

PGM (Program)

Specifies the qualified name by which the compiled program is known.

<u>*CURLIB:</u> If a library is not specified, the program is created in the current library. If no current library entry exists in the library list, QGPL is used.

library-name: Specify the name of the library where the compiled program is created.

Warning: If the program name you specify is the same name of an existing program, your new program replaces the existing one.

program-name: Specify the name of the program being created that contains the SQL statements.

Optional Parameters

The following parameters are optional for the SQL precompiler source commands. If you choose not to specify any of the following keywords or their values, then the defaults are used.

SRCFILE (Source File)

Specifies the qualified name of the source file that contains the source with the SQL statements.

***LIBL:** Specifies that the library list is used to locate the source file.

***CURLIB:** Specifies that the current library for the job is used to locate the source file. If no current library entry exists in the library list, QGPL is used.

library-name: Specify the library where the source file is located.

QxxxSRC: If the source file name is not specified, the IBM-supplied source file name is used. QLBLSRC is the source file name for COBOL, QPLISRC for PL/I, QRPGSRC for RPG, and QCSRC for C.

source-file-name: Specify the name of the source file that contains the source. This source file should have a record length of 92. The source file can be a database file, device file, or an inline data file.

SRCMBR (Source Member)

Specifies the name of the source file member containing the source. This parameter is only specified if the source file name in the SRCFILE parameter is a database file.

***PGM:** Specifies that the host program source is in the member of the source file that has the same member name as that specified in the PGM parameter for the precompiler command.

source-file-member-name: Specify the name of the member containing the host program source.

OPTION

Specifies the following options to the precompiler. If an option is specified more than once, or if two options conflict, the last option specified is used. If an option is not specified, then the default is used.

***NOSOURCE or *NOSRC:** Specifies that a source listing is not produced by the precompiler.

***SOURCE or *SRC:** Specifies that a source listing is produced by the precompiler, consisting of the source input and all error messages.

***NOXREF:** Specifies that the precompiler does not produce a cross-reference of names.

***XREF:** Specifies that the precompiler produces a cross-reference between items in your program and the numbers of the statements in your program that refer to these items.

***GEN:** Specifies that the host language compiler is to be called to create a program after the source has been precompiled.

***NOGEN:** Specifies that the host language compiler is not called and that no program will be created.

***QUOTESQL (COBOL only):** Specifies that the string delimiters within SQL statements are quotation marks (").

***APOSTSQL (COBOL only):** Specifies that the string delimiters within SQL statements are apostrophes (').

Notes:

1. If *APOSTSQL is specified, the SQL escape character for delimited identifiers is the quotation (") mark. If *QUOTESQL is specified, the SQL escape character for delimited identifiers is the apostrophe (').
2. SQL statements in RPG, PL/I, and C use apostrophes for string delimiters and quotation marks for SQL escape characters.

***QUOTE (COBOL only):** Specifies that a quotation mark (") is used for nonnumeric literals and Boolean literals in COBOL statements.

***APOST (COBOL only):** Specifies that an apostrophe (') is used for non-numeric literals and Boolean literals in COBOL statements.

***SYSVAL (COBOL and RPG only):** Specifies that the value used as the decimal point is from the QDECFMT system value.

Note: If the QDECFMT system value specifies that the value used as the decimal point is a comma, any numeric constants in lists (such as in the SELECT clause, VALUES clause, and so on) must be separated by a comma followed by a blank. For example, VALUES(1,1, 2,23, 4,1) is equivalent to VALUES(1.1,2.23,4,1) where the decimal point is the period.

***PERIOD (COBOL and RPG only):** Specifies that the value used as the decimal point is a period.

Note: In PL/I and C, the period is used as the decimal point.

***COMMA (COBOL and RPG only):** Specifies that the value used as the decimal point is a comma.

Note: Any numeric constants in lists (such as in the SELECT clause, VALUES clause, and so on) must be separated by a comma followed by a blank. For example, VALUES(1,1, 2,23, 4,1) is equivalent to VALUES(1.1,2.23,4,1) where the decimal point is the period.

***SYS:** Specifies that the AS/400 system naming convention will be used, (library-name/file-name).

***SQL:** Specifies that the SQL naming convention will be used (collection-name.table-name).

Note: *SYS also specifies that the AS/400 security convention will be used. *SQL specifies that the SQL security convention will be used. The AS/400 security convention requires that the user running the program must have authority to objects referred to by the SQL statements in the program. The SQL security convention requires that the owner of the program has authority to the objects referred to in the static SQL statements in the program. The SQL security convention is the same as the AS/400 security convention for dynamic SQL and interactive SQL.

TGTRLS (target release-RPG and COBOL only)

Specifies whether the precompiler and compiler are to check statements for potential restore onto the prior release of the OS/400 program.

The *PRV option must be used if the program is to be restored onto the prior OS/400 program.

***CURRENT:** No checking is required.

***PRV:** The SQL precompiler will issue message SQL7906. This message states that the program must not be restored to an OS/400 Release 1 Modification Level 0 system. It can be restored to an OS/400 Release 1 Modification Level 2 or Release 2 Modification Level 0 system.

The user must also save the object with *PRV to run it on the previous release.

For more information, see the *PRV option descriptions for the TGTRLS parameter on the Create RPG Program (CRTRPGPGM) and Create COBOL Program (CRTCBLPGM) commands in the *CL Reference* manual.

INCFILE (Include File)

Specifies the qualified name of the source file that contains the members included in the program with any SQL INCLUDE statement.

***LIBL:** Specifies that the library list is used to locate the source file.

***CURLIB:** The current library for the job used to locate the source file. If no current library entry exists in the library list, QGPL is used.

library-name: Specifies the library where the source file is located.

***SRCFILE:** Specifies the qualified source file you specified in the SRCFILE parameter that contains the source file member(s) specified on any SQL INCLUDE statement.

source-file-name: Specify the name of the source file that contains the source file member(s) specified on any SQL INCLUDE statement. The record length of the source file you specify here must be no less than the record length of the source file you specified for the SRCFILE parameter.

COMMIT

Specifies if SQL statements in the compiled program are run under commitment control. Files referred to in the host language source are not affected by this option. Only files, SQL tables, and SQL views referred to in SQL statements are affected.

***CHG (change):** Specifies that only the updated, deleted, and inserted rows are locked until the end of the unit of recovery (transaction).

***ALL:** Specifies that all rows selected, updated, deleted, and inserted are locked until the end of the unit of recovery (transaction).

***NONE:** Specifies that commitment control is not used. If SQL DDL statements are included in the program, *NONE must be used.

Note: If *CHG or *ALL are specified, the following data definition statements cannot be included in the application:

COMMENT ON	DROP
CREATE COLLECTION	GRANT
CREATE INDEX	LABEL ON
CREATE TABLE	REVOKE
CREATE VIEW	

GENLVL (generation level)

Specifies if a program is created, depending on the severity of messages generated as a result of precompile errors. If errors occur with a severity level greater than the value specified in GENLVL parameter, the appropriate language compiler is not called.

10: If a severity level value is not specified, the default severity level is 10.

severity-level: Specify a number from 0 through 40. Some suggested values are listed below:

10 The level value for warnings.
20 The level value for general error messages.
30 The level value for serious error messages.
40 The level value for system detected error messages.

Note: The GENLVL only applies to messages generated as a result of SQL precompile errors. The specified GENLVL value is **not** passed to the language compiler.

MARGINS (PL/I and C only)
Specifies the part of the precompiler input record that contains source text.

***SRCFILE:** Specifies that the margin values of the file member you specify in the SRCMBR parameter are used. For PL/I, if the file is type SQLPLI, the margin values are the values specified on the Source Entry Utility Services display; if the file is a different type, the margin values are the default values of 2 and 72. For C, if the file is type SQLC, the margin values are the values specified on the Source Entry Utility Services display; if the file is a different type, the margins are the default values of 1 and 80.

left right: Specify the beginning position (**left**) and the ending position (**right**) for the statements. The margins you specify must not be less than 1 or more than 80. The left margin must be smaller than the right margin.

PRTFILE (printer file)
Specifies the qualified name of the printer device file to which the precompiler listing is directed. The file should have a minimum length of 132 characters. If a file with a record length of less than 132 characters is specified, information is lost.

***LIBL:** Specifies the library list used to locate the printer file.

***CURLIB:** Specifies the current library for the job used to locate the printer file. If no current library entry exists in the library list, QGPL is used.

library-name: Specify the library where the printer file is located.

QSYSPRT: If a file name is not specified, the precompiler listing is directed to the IBM-supplied print file, QSYSPRT.

printer-file-name: Specify the name of the printer device file to which the precompiler listing is directed.

TEXT
Specifies text that briefly describes the program and its function.

***SRCMBRTXT:** Specifies that the text is taken from the source file member being used to create the program. You can add to or change the text for a database source member by using the Start Source Entry Utility (STRSEU) command, or by using either the Add Physical File Member (ADDPFM) or Change Physical File Member (CHGPFM) command. If the source file is an inline file or a device file, the text is blank.

***BLANK:** Specifies no text.

description: Specify no more than 50 characters enclosed in apostrophes (').

Example of the Precompiler Source Command

The following example creates a COBOL program named SAMPLE and stores it in a library named QTEMP. The SQL naming convention was selected, and every row selected from the specified table(s) are locked until the end of the unit of recovery.

```
CRTSQLCBL PGM(qtemp/sample) OPTION(*SRC *XREF *SQL) COMMIT(*ALL)
```

Compiling an Application Program

The SQL/400 program automatically calls the host language compiler after the successful completion of a precompile, unless *NOGEN is specified. The CRTxxxPGM command is run specifying the program name, source file name, precompiler created source member name, text, and USRPRF. For COBOL and RPG, the TGTRLS parameter from the CRTSQLxxx command is specified on the CRTxxxPGM command. For COBOL, the *QUOTE or *APOST is also passed. For PL/I and C, the MARGINS are set in the temporary source file. Defaults are used for all other parameters.

Note: If you specify *SYS, USRPRF(*USER) is specified by the precompiler. If you specify *SQL, USRPRF(*OWNER) is specified by the precompiler.

You can interrupt the call to the host language compiler by specifying *NOGEN under OPTION parameter of the precompiler command. *NOGEN specifies that the host language compiler will not be called. The precompiler has created the source member in the QTEMP/QSQLTEMP file, using the program name specified in the CRTSQLxxx command as the name of the member. You now can explicitly call the host language compilers, specifying the source member in the QTEMP/QSQLTEMP file, and change the defaults, if you wish.

If you precompiled using the *SQL option, then you must specify USRPRF(*OWNER) on the CRTxxxPGM command.

Note: You must not change the source member in QTEMP/QSQLTEMP prior to issuing the CRTxxxPGM command or the compile will fail.

Warning: If you do separate precompile and compile steps, and the source program contains references to externally described files (COPY DDS in COBOL, %INCLUDE in PL/I, and externally defined files or data structures in RPG), the referred to files must not be changed between the precompile and compile steps. Otherwise, results that are not predictable may occur because the change to the field definitions are not reflected in the temporary source member.

Error and Warning Messages during a Compile

The conditions described in the following paragraphs could produce an error or warning message during an attempted compile process.

During a PL/I or C Compile: If EXEC SQL starts before the left margin (as specified with the MARGINS parameter, the default), the SQL precompiler will not recognize the statement as an SQL statement. Consequently, it will be passed as is to the compiler.

During a COBOL Compile: If EXEC SQL starts before column 12, the SQL precompiler will not recognize the statement as an SQL statement. Consequently, it will be passed as is to the compiler.

Binding an Application

Before you can run your application program, a relationship between the program and any referred to tables and views must be established. This process is called **binding**. The result of binding is an **access plan**.

The access plan is a control structure that describes the actions necessary to satisfy each SQL request. An access plan contains information about the program and about the data the program intends to use.

The access plan is stored in the program and therefore is automatically deleted, moved, saved, and so on with the program.

SQL automatically attempts to bind and create access plans when a successful compile has occurred. If, at run time, the database manager detects that an access plan is not valid or detects that changes have occurred to the database that may improve performance (for example, the addition of indexes), a new access plan is automatically created. Binding does three things:

1. **It revalidates the SQL statements using the description in the database.** During the bind process, the SQL statements are checked for valid table, view, and column names. If a referred to table or view does not exist at the time of the precompile or compile, the validation is done at run time. If the table or view does not exist at run time, a negative SQLCODE is returned.
2. **It selects the access paths needed to access the data your program wants to process.** In selecting an access path, indexes, table sizes, and other factors are considered, when it builds an access plan. It considers all indexes available to access the data and decides which ones (if any) to use when selecting a path to the data.
3. **It attempts to build access plans.** If all the SQL statements are valid, the bind process then builds and stores access plans in the program.

If the characteristics of a table or view your program accesses have changed, the access plan may no longer be valid. When you attempt to run a program that contains an access plan that is not valid, the system automatically attempts to rebuild the access plan. If the access plan cannot be rebuilt, a negative SQLCODE is returned. In this case, you might have to change the program's SQL statements and reissue the CRTSQLxxx command to correct the situation.

For example, if a program contains an SQL statement that references COLUMNA in TABLEA and the user deletes and recreates TABLEA so that COLUMNA no longer exists, when the user calls the program, the automatic rebind will be unsuccessful because COLUMNA no longer exists. In this case the user must change the program source and reissue the CRTSQLxxx command.

Program References

All references to collections, tables, views, and indexes in an SQL program are placed in the object information repository (OIR) of the library when the program is created.

You may use the Display Program References (DSPPGMREF[2]) CL command to display all object references in the program. If the SQL naming convention is used, the library name is stored in the OIR in one of two ways:

1. If the SQL name is fully qualified, the collection name is stored.
2. If the SQL name is not fully qualified, the authorization ID of the statement is stored.

If the system naming convention is used, the library name is stored in the OIR in one of two ways:

1. If the object name is fully qualified, the library name is stored.
2. If the object is not fully qualified, *LIBL is stored.

Running a Program with Embedded SQL

Running a host language program with embedded SQL statements, after the precompile and compile have been successfully done, is the same as running any host program. Type:

```
CALL pgm-name
```

on the system command line. For more information on running programs, see the *CL Programmer's Guide*.

OS/400 DDM Considerations

SQL does not support remote file access through OS/400 DDM (distributed data management).

Override Considerations

You can use overrides (specified by the OVRDBF command) to direct a reference to a different table or view or change certain operational characteristics of the program. The following parameters are processed if an override is specified:

TOFILE
MBR
SEQONLY
LVLCHK
INHWRT
WAITRCD

All other override parameters are ignored. For more information on overrides, see the *Database Guide* and the *Data Management Guide*.

SQL Return Codes

A list of SQL return codes is provided in Appendix B.

2 For more information about the DSPPGMREF CL command, see the *CL Reference* manual.

Chapter 11. Using Interactive SQL

This chapter describes how to use interactive SQL to syntax check, validate, run SQL statements, and use the prompt function. An overview, functional description, and tips on using interactive SQL are provided.

Overview

Interactive SQL allows you to enter SQL statements or prompts directly from the keyboard. Either a completion message or an error message is displayed after each statement is processed. In addition, status messages are normally displayed during long running statements. In the case of the SELECT statement, the result is also displayed, printed, or sent to a database file.

There are four basic functions supplied by interactive SQL:

- The **statement entry** function allows you to type the interactive SQL statement and run it by pressing the Enter key.

  While in the statement entry function, you may:

  - Type in an interactive SQL statement and run it.
  - Page through previous statements and messages.
  - Call session services.
  - Call list selection.
  - Edit statements.
  - Prompt for SQL statements.

- The **prompt** function allows you to type an SQL statement, press F4 (Prompt) and be prompted for the syntax of the statement. It also allows you to press F4 to get a menu of all SQL statements. From this menu you can select a statement and be prompted for the syntax of the statement.

- The **session services** function allows you to:

  - Change commitment control attributes.
  - Change the values that are in effect when you run your interactive SQL statements.
  - Change the SELECT output device.
  - Change the list of collections (libraries).
  - Change the list type to select either all your system and SQL objects or only your SQL objects.
  - Print the current session history.
  - Remove all entries you made for the current session.
  - Save the session in a source file.
  - Change the data refresh operation when displaying data.

- The **list selection** function allows you to select from lists of your authorized collections, tables, views, or columns. The selections you make from the lists can be inserted into the SQL statement at the position located by the cursor.

Online help is available for all the displays in interactive SQL, as well as other normal system display operations.

If your previous SQL session was saved or ended abnormally, the first display that appears when you start a new SQL session is the Recover SQL Session display. The Recover SQL Session display allows you to continue with the old session or

© Copyright IBM Corp. 1988, 1989

start a new session. If you do not recover the old session at this time, the old session is deleted.

You cannot run all of the SQL statements in the interactive SQL environment. For a list of valid interactive statements, see the "Supported SQL Statements" on page 11-10. However, you can use interactive SQL to check the syntax of all the SQL statements in multiple languages.

The following sections describe in more detail the functions available.

Terminology

There are two naming conventions that can be used on SQL/400: system (*SYS) and SQL (*SQL). The naming convention used affects the terms used on the displays and the method for qualifying file and table names. In the system naming convention, files are qualified by library name in the form: library/file. In the SQL naming convention, tables are qualified by the collection name in the form: collection.table. The naming convention used is selected by a parameter on the Start SQL (STRSQL) command[1] on a command line. In the following discussion, the SQL naming convention is used.

The following list shows the relationship between AS/400 object names and SQL/400 object names:

System Terms	SQL Terms
Library	Collection
Physical file	Table
Record	Row
Field	Column
Logical file	View

For further descriptions of the terms listed above, see the section "Relational Databases and Terminology" on page 1-2.

[1] For a description of the STRSQL command, see the "STRSQL Command" on page 11-18.

Getting Started

You can start using interactive SQL by entering STRSQL[1] on a command line. The Enter SQL Statements display then appears. This is the primary display from which you can enter SQL statements and, by using the function keys, use the session services (F13), list select functions (F16=Select collections, F17=Select tables, or F18=Select columns), and prompting (F4).

```
                          Enter SQL Statements

 Type SQL statement, press Enter.
 ===>____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
                                                              Bottom
  F3=Exit   F4=Prompt   F6=Insert line   F9=Retrieve   F10=Copy line
  F12=Cancel            F13=Services     F24=More keys
                                         (C) COPYRIGHT IBM CORP. 1982, 1989.
```

Press F24 to view function keys F16, F17, and F18.

```
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
     ____________________________________________________________
                                                              Bottom

  F14=Delete line   F15=Split line     F16=Select collections (libraries)
  F17=Select tables                    F18=Select columns   F24=More keys
           (files)                               (fields)
```

Note: The names in parentheses appear in place of the ones above, if you select the system naming convention.

Audience

The intended users of interactive SQL are programmers and database administrators with a knowledge of database concepts and the SQL language.

Functional Description

Interactive SQL allows the programmer or database administrator to quickly and easily define, update, delete, or look at data, mainly for testing, problem analysis, and database maintenance. A programmer, for example, may insert some rows into a table, using interactive SQL, and test the SQL statements before running them in an application program. A database administrator may use interactive SQL to grant or revoke privileges, create or drop collections, tables, or views, or select information from system catalog tables.

Statement Entry

The statement entry function is the function that you are in when first entering interactive SQL. You return to the statement entry function after processing each interactive SQL statement.

In the statement entry function, you type or prompt for the entire SQL statement and submit it for processing by pressing the Enter key.

The statement may be one or many lines long. When the statement has been processed, the statement and the resulting text message are moved upward on the display. You can then enter another statement.

If the statement was recognized by SQL but contains a syntax error, the statement that was just entered and the resulting text message (syntax error) is moved upward on the display, and a copy of the statement with the error appears in the input area with the cursor positioned at the beginning of the token that contains the syntax error.

You may page through previous statements, commands and messages—even those that have moved off the top of the display. If you need more room to enter an SQL statement, you can page down the display.

Prompting

The prompt function helps you supply the necessary information for the syntax of a statement you want to use. The prompter is useful when you have forgotten the syntax of a statement or when you want to save time.

To use the prompter, you have two options:

- If you type at least the verb of the statement before pressing F4 (Prompt), any (partial) statement beyond the verb, in which the syntax is correct, will be accepted. The statement will be parsed and the clauses that have been completed will be *filled in* on the prompt displays.

  There are menus available if you enter an unqualified CREATE, DECLARE or DROP statement (without the noun).

- If you press F4 (Prompt) without typing anything on the Enter SQL Statements panel, you are shown a complete menu of statements. You can select the number of the statement you want to use, and the system will then prompt for the statement you selected.

  Note: In a few cases, if you type part of an SQL statement in the correct syntax, and the last item in the partial statement is an object name that is too long, the prompt screen will appear and the object name that is too long will be truncated to the allowable maximum length.

The prompt function can be used in any of the three run modes (*RUN, *VLD, and *SYN).

The statement that is built as a result of the prompting will be inserted into the session. If you cancel prompting using F12 or F3, a message will be inserted indicating that you cancelled prompting.

The prompter remains in control if any error is encountered. The cursor will be positioned to the error whenever possible.

Statements are checked for syntax upon entry into the prompter. A *syntax check* here means that the statement is syntactically correct so far. The prompter will not accept a syntactically incorrect statement. You can use F12 or F3 to exit or cancel whether or not the statement is syntactically correct.

Notes:

1. In *RUN or *VLD mode, only the interactive form of the data manipulation statement is allowed. In *SYN mode with a language other than *NONE, all SQL statements may be prompted.
2. In SQL (*SQL) naming, a default collection (library) name may be supplied when the collection name qualifies a table, view, or index name. The default used depends on the value of the LIBOPT parameter specified on the STRSQL command, (option 4 (Session services)). If the value of LIBOPT is *CURLIB, then the current library name is used as the default. If the value of LIBOPT was a user-specified collection name, then that collection name is used as the default. For all other values of LIBOPT, the default collection name is the current job's profile name (sign-on ID).

   The default supplied on the prompt displays may be different from the implicit qualification used when no collection name is supplied on the Enter SQL Statements display. On the Enter SQL Statements display, the implicit qualification is always the current job's profile name.
3. In system (*SYS) naming, unqualified collection names are qualified by *LIBL.

If you are in the SQL prompter and press F4 (Prompt), a list of objects will be displayed, depending on the position of the cursor. You can select objects from this list. If you press F21 (Display statement), the prompter will display a formatted SQL statement as it has been *filled in* so far.

DBCS Considerations for Prompting: The rules for processing DBCS data across multiple lines, in such a way as to allow entering one continuous DBCS character string, are the same as those adopted by the Enter SQL Statements display. Each line must contain as many shift-in characters as shift-out characters. However, if during formatting, the prompter finds that the very last column on a line contains a shift-in and the very first column of the next line contains a shift-out, the shift-in and shift-out characters are removed by the prompter when the two lines are assembled. Also, if the last two columns of a line contain a shift-in followed by a single-byte blank, and the very first column of the next line contains a shift-out, the shift-in, blank, shift-out sequence is removed by the prompter when the two lines are assembled.

There is an unavoidable situation that may come up as a result of following the rules above. This situation is a result of a statement code assembled by the prompter that may align itself on boundaries when passed back to the statement entry panel in such a way that the statement entry code will then strip shift-in and shift-out pairs

unknown to the user when (if) the statement is run or verified. This condition will be sensed by the prompter before the statement is processed and returned as a warning message. As far as the prompter is concerned, the statement is syntactically correct when it is leaving the prompter, but the prompter is aware during processing that the condition exists for unexpected results and supplies the message.

You can either ignore this message or take further action. Interactive SQL is sensitive to this situation and (in this case only) shows the statement to you before it is run or verified (along with the message). Message help text describes the situation that has occurred, discusses how the statement will be processed, and indicates how to proceed.

Session Services

The session services function is made available by pressing F13 (Session services) on the Enter SQL Statements display, which accesses the Work with Session Services display. It is through this function that you can change the current values that are in effect for your interactive SQL session. You also can print, clear, or save the session to a source file. The following is a brief description of the options on the Work with Session Services display:

1. Change commitment control

   This function lets you set the level of commitment control that you want in effect for your session. The choices are *NONE, *CHG, and *ALL.

   Note: If locks on rows are currently being held for this unit of recovery and you attempt to change COMMIT from *ALL or *CHG to *NONE, a warning message is displayed.

   For more information on the COMMIT and ROLLBACK statements, see the *SQL/400 Reference* manual.

2. Change statement processing control

   This function lets you control how your SQL statement is processed. You can do one of the following:

   a. Check for correct syntax, make a validity check, and run your SQL statement (the Run option).
   b. Check for correct syntax and check that the object names used in the statement actually exist on your system (called a validity check), but not run the statement. This option does not allow the use of host variables.
   c. Check for correct syntax only. This option allows you to use host variables and allows *all* SQL statements.

      When this option is chosen, you will be prompted to select what language character set to use for syntax checking SQL statements. Valid options are:
      - *NONE
      - *C
      - *CBL
      - *PLI
      - *RPG

      If you choose *CBL or *RPG, you are prompted for the decimal point symbol to use. The choices are the system value (default), period, or comma. For interactive SQL, the decimal point is set to the system value; for PL/I and C, the decimal point is set to a period.

      Additionally, with *CBL, you will be prompted for the SQL string delimiter symbol you want to use. The choices are quote (default) and apostrophe.

For the other languages, this value is set to an apostrophe and cannot be changed.

When you run the statement, a syntax check and validity check are performed automatically.

3. Change SELECT output device

   This function lets you control where the output from a successful SELECT statement is sent. You can either display, print, or save the results in a database file. If you choose to print or save the results, you are prompted for printer or database file information.

4. Change list of collections (libraries)

   This function lets you control which collections and libraries are used as a basis for building the collections list when F16 is pressed. You can select from *LIBL, *USRLIBL, *ALLUSR, *ALL, and *CURLIB, or specify a library name. The name you specify may be used to supply a single collection or library. You are prompted for the name. For more information on the other five choices, see the *CL Reference* manual.

   When you change the collection (library) option, it affects the presentation of the lists only and not the statement syntax. In the system naming convention, the *LIBL list is used to resolve an unqualified file name, regardless of the setting of the list of collections or of the STRSQL LIBOPT parameter. In the SQL naming convention, the user profile is used to resolve an unqualified table name. If there is a collection with the same name as your user profile, that collection name is used to resolve the unqualified table, regardless of the setting of the list of collections or of the STRSQL LIBOPT parameter.

   In either naming convention, selecting the file or table name from the list ensures that the name is properly qualified.

5. Change list type

   This function lets you control which types of objects are shown when you request a list. You may choose to see both system created objects and SQL created objects, or SQL created objects only.

6. Print current session

   This function lets you print the current session immediately and then continue working. You are prompted for printer information. All the SQL statements you entered and all the messages displayed are printed just as they appear on the Enter SQL Statements display.

7. Remove all entries from current session

   This function lets you remove all the SQL statements and messages from the Enter SQL Statements display and the session history. You are prompted to ensure that you really want to delete the information.

8. Save session in source file

   This function lets you save the session in a source file. You are prompted for the source file name. This function lets you embed the source file into a host language program by using the source entry utility (SEU).

9. Change data refresh option

   This option lets you control the data refresh operation when using the interactive SELECT statement to display data. You are prompted to specify whether the data is always refreshed or whether the data is only refreshed on the first forward pass of the retrieval operation.

List Selection Function

The list selection function is made available by pressing F16, F17, or F18 on the Enter SQL Statements display. After pressing the desired function key, you are presented with a list of authorized collections, tables, and views, or columns from which to choose. If you request a list of tables, but you have not previously selected a collection, you are asked to select a collection first. If you request a list of columns, but you have not previously selected a table and/or a collection, you are asked to select a collection and/or table first.

The list selection mode allows you to do the following:

- Select one or more items from the list, numerically specifying the order.
- Page through the list.
- Display the list again (all input fields are erased).
- Obtain help.
- Exit the list selection function without making any choices.

When returning from the list function, the selections you made are inserted at the position of the cursor on the Enter SQL Statements display. When using the table and view list to insert names into you statement, a maximum of 32 tables and/or views may be selected from the list. For more tips on using the list selection function, see "Using the List Selection Function" on page 11-14.

Note: **An important rule for using the list function:** Always select the list you are primarily interested in. For example, if you want a list of columns, but you believe that the columns you want are in a collection or table not currently selected, press F18 (Select columns). Then, from the column list, press the appropriate function key to change the table or collection. Do not go first to the collection or table list, in this case, because you do not want to have the table name or collection name inserted into your statement.

For more information on using the list selection function, see "Using the List Selection Function" on page 11-14.

Exit Interactive SQL

Pressing F3 (Exit) allows you to exit the interactive SQL environment and do one of the following:

1. Save and exit session; leave interactive SQL.
2. Exit without saving session; leave interactive SQL without saving your session.
3. Resume session; remain in interactive SQL and return to the Enter SQL Statements display (the current session parameters remain in effect).
4. Save session in source file; save the current session in a source file as defined in the Change Source File display when Enter is pressed.

Note: If locks on rows are currently being held for this unit of recovery and you attempt to exit interactive SQL, a warning message is displayed.

Help

Help is available from every display in the interactive SQL session. You can obtain general help information about the whole display, or selective information about part of the displayed information by positioning the cursor in the area in question and pressing the Help key or F1 (Help).

The Session and Its Functions

When you enter the start SQL command (STRSQL)[2], an interactive SQL session is created. The session consists of:

- Values of parameters you specified in the STRSQL command
- SQL statements you entered in the session along with their corresponding messages following each SQL statement
- Values of the parameters that you changed via the session services function, if applicable
- Lists selections you have made

Interactive SQL supplies a unique session-id. This allows multiple users with the same sign-on ID to use interactive SQL from more than one work station at the same time. Also, more than one interactive SQL session can be run from the same work station at the same time by the same user (sign-on ID).

Recovering a Saved or Failed SQL Session

If the previous SQL session was saved or ended abnormally, interactive SQL presents the Recover SQL Session display at the start of the next session (when the next STRSQL command is entered). From this display, you can either recover the old session by selecting option 1 (Attempt to resume existing SQL session) or delete the old session and start a new session by selecting option 2 (Delete existing SQL session and start a new session). If you choose to recover the old session, the parameters you specified when you entered STRSQL are ignored and the parameters for the old session are used. If you choose to delete the old session and continue with the new session, the parameters you specified when you entered STRSQL are used.

Messages

Only first-level text for messages are shown on the Enter SQL Statements display. You can see the second-level text by positioning the cursor to the first-level text and pressing the Help key.

If an error is detected in a SQL statement, the statement is duplicated and the cursor is positioned at the place in the statement where the error was detected.

2 For a description of the STRSQL command, see the "STRSQL Command" on page 11-18.

Supported SQL Statements

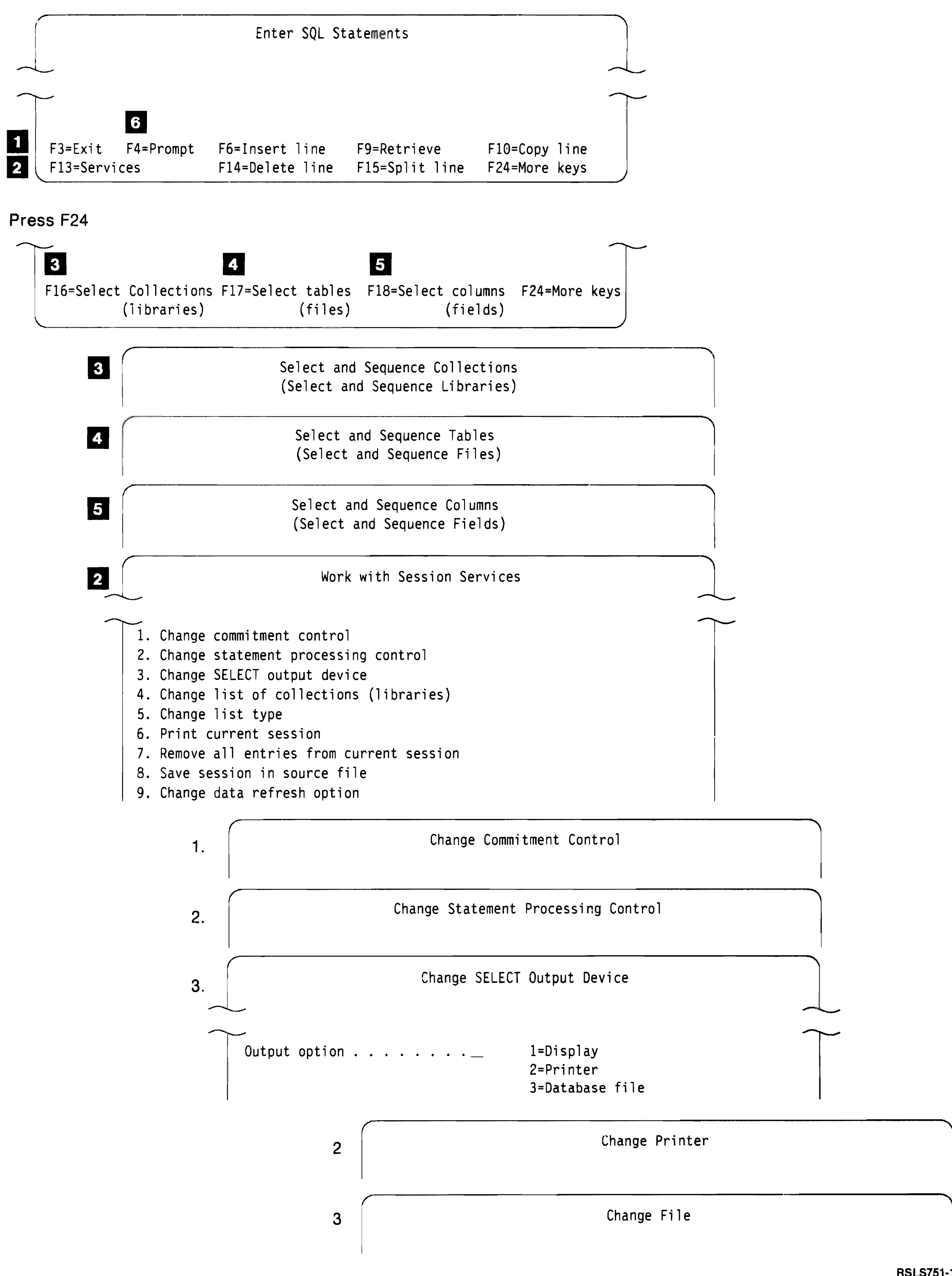
The following SQL statements can be run in the interactive SQL environment. All SQL statements can be syntax checked by interactive SQL. See the *SQL/400 Reference* manual for information about the statements.

In the case where there is a distinction between the interactive form of the SELECT statement and other forms, the interactive form must be used.

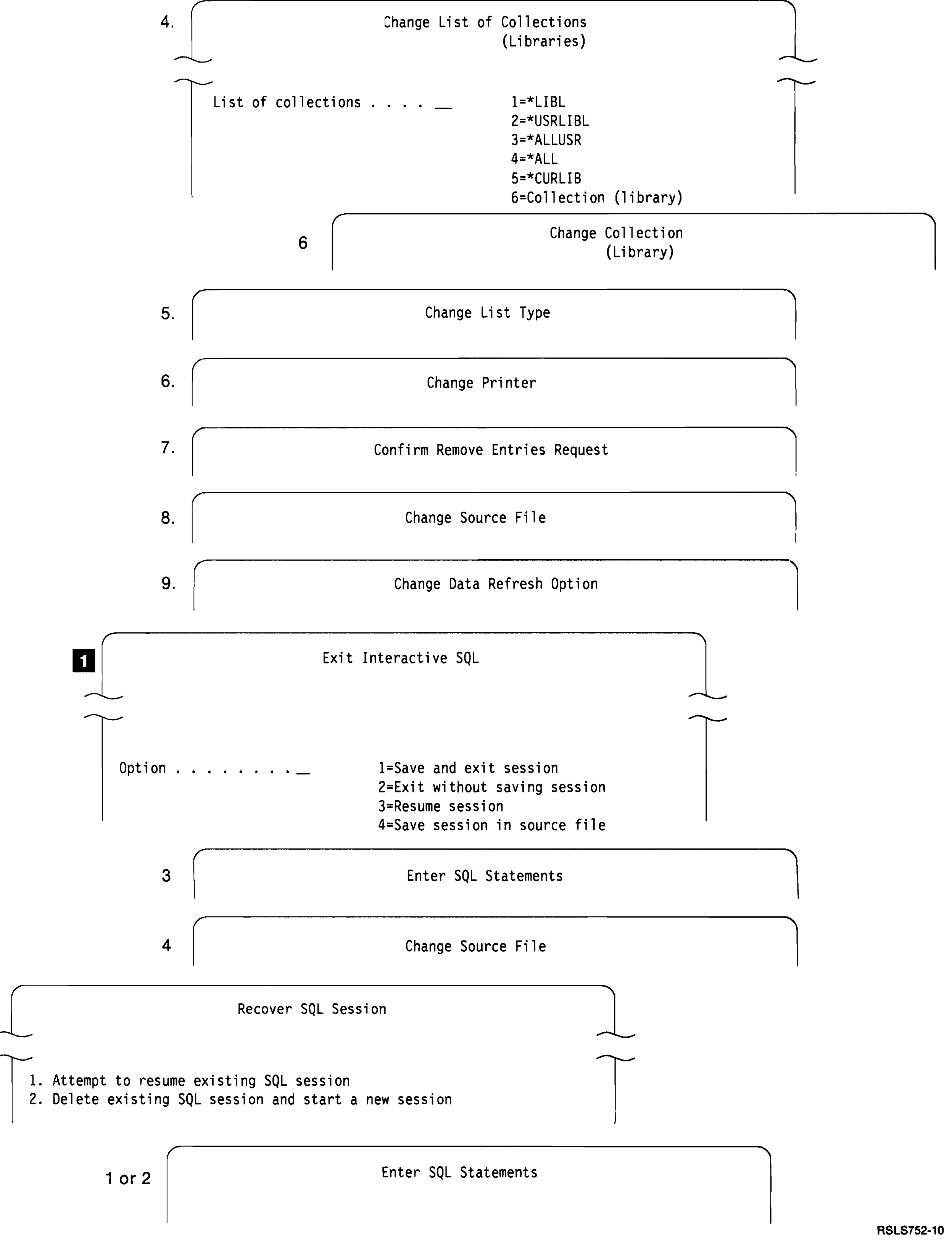
COMMENT ON	GRANT
COMMIT	INSERT
CREATE COLLECTION	LABEL ON
CREATE INDEX	LOCK TABLE
CREATE TABLE	REVOKE
CREATE VIEW	ROLLBACK
DELETE	SELECT
DROP	UPDATE

Interactive Session Display Flow Diagram

The following figure shows the overall flow of the displays used in an interactive SQL session. The reverse characters are used to represent the function key pressed to see the next display in the figure. Remember, the only statement that returns data is the SELECT statement.



Enter SQL Statements
6
1 F3=Exit F4=Prompt F6=Insert line F9=Retrieve F10=Copy line
2 F13=Services F14=Delete line F15=Split line F24=More keys
Press F24
3 4 5
F16=Select Collections (libraries) F17=Select tables (files) F18=Select columns (fields) F24=More keys
3 Select and Sequence Collections (Select and Sequence Libraries)
4 Select and Sequence Tables (Select and Sequence Files)
5 Select and Sequence Columns (Select and Sequence Fields)
2 Work with Session Services
1. Change commitment control
2. Change statement processing control
3. Change SELECT output device
4. Change list of collections (libraries)
5. Change list type
6. Print current session
7. Remove all entries from current session
8. Save session in source file
9. Change data refresh option
1. Change Commitment Control
2. Change Statement Processing Control
3. Change SELECT Output Device
Output option . . . . _
1=Display
2=Printer
3=Database file
2 Change Printer
3 Change File
RSLS751-10


4.
Change List of Collections
(Libraries)
List of collections . . . . __
1=*LIBL
2=*USRLIBL
3=*ALLUSR
4=*ALL
5=*CURLIB
6=Collection (library)
6
Change Collection
(Library)
5.
Change List Type
6.
Change Printer
7.
Confirm Remove Entries Request
8.
Change Source File
9.
Change Data Refresh Option
1
Exit Interactive SQL
Option . . . . __
1=Save and exit session
2=Exit without saving session
3=Resume session
4=Save session in source file
3
Enter SQL Statements
4
Change Source File
Recover SQL Session
1. Attempt to resume existing SQL session
2. Delete existing SQL session and start a new session
1 or 2
Enter SQL Statements
RSLS752-10

For example, press F4 to prompt without typing anything:

6

```
                         Select SQL Statement

Select one of the following:

     1. COMMENT ON
     2. COMMIT
     3. CREATE COLLECTION
     4. CREATE INDEX
     5. CREATE TABLE
     6. CREATE VIEW
     7. DELETE
     8. DROP COLLECTION
     9. DROP INDEX
    10. DROP TABLE
    11. DROP VIEW
    12. GRANT
    13. INSERT
    14. LABEL ON
```

For example, type CREATE TABLE and press F4 to prompt:

6

```
                    Specify CREATE TABLE Statement

Type choices.

  Table . . . . . . . . . . . . .        __________  Name
    Collection . . . . . . . . . .        __________ Name, F4 for list

Type information, press Enter.
  Default:   Y=Yes, N=No
  Data:      1=BIT, 2=SBCS, 3=MIXED

                                  -----Digits-----
Column        Type                Precision  Scale  Length  Default  Data
__________    ______________         __        __    _____     Y       2
```

RSLS788-2

Tips on Using Interactive SQL

The following paragraphs contain tips and techniques for using interactive SQL.

Using the List Selection Function

You may request a list at any time while typing an SQL statement on the Enter SQL Statements display. The selections you make from the lists are inserted on the Enter SQL Statements display, starting where the cursor is located and in the order that you specify. You must type the key words of the statement, but the selected list information is added automatically.

The examples on the following pages show you how to use the list select function to build your SELECT statement.

Note: The following examples let you see how you might obtain the specific data you desire. The object names and selections are imaginary and probably do not exist on your system.

First Time Used in Session

First, assume you have *just entered* interactive SQL by typing STRSQL on the system Command Entry display, and you have made no list selections or entries. Also, assume you have selected *SQL for the naming convention.

At the Enter SQL Statements display:

1. Type SELECT on the first data entry line.
2. Type FROM on the second data entry line.
3. Leave the cursor positioned after FROM.

```
                              Enter SQL Statements

Type SQL statement, press Enter.
===> SELECT
     FROM _
```

4. Press F17 (Select tables) to obtain a list of tables, because you want the table name to follow FROM.

   Note: Instead of a list of tables appearing as you expected, a list of collections appears (the Select Collections display). This is because you have not selected a collection or collections from which to work, since you just entered the SQL session.

5. Type a 1 in the *Seq* column across from YOURDB2 collection.

```
                         Select and Sequence Collections


Type sequence numbers (1-999) to select collections, press Enter.

Seq   Collection      Type   Text
        YOURDB1       SYS    Company benefits
 1      YOURDB2       SYS    Employee personal data
        YOURDB3       SYS    Job classifications/requirements
        YOURDB4       SYS    Company insurances
```

6. Press the Enter key.

   The Select and Sequence display appears, showing the tables existing in the YOURDB2 collection.

7. Type a 1 in the *Seq* column across from PEOPLE table.

```
                    Select and Sequence Tables

Type sequence numbers (1-999) to select tables, press Enter.

Seq    Table/view   Collection     Type   Text
        EMPLCO       YOURDB2       TAB    Employee company data
 1      PEOPLE       YOURDB2       TAB    Employee personal data
        EMPLEXP      YOURDB2       TAB    Employee experience
        EMPLEVL      YOURDB2       TAB    Employee evaluation reports
        EMPLBEN      YOURDB2       TAB    Employee benefits record
        EMPLMED      YOURDB2       TAB    Employee medical record
        EMPLINVST    YOURDB2       TAB    Employee investments record
```

8. Press the Enter key.

   The Enter SQL Statements display appears again with the table name, YOURDB2.PEOPLE, inserted after FROM.

   The table name is qualified by the collection name, and both names appear in the proper syntax.

```
                           Enter SQL Statements

Type SQL statement, press Enter.
===> SELECT
     FROM YOURDB2.PEOPLE _
```

9. Position the cursor after SELECT.

10. Press F18 (Select and Sequence columns) to obtain a list of columns, because you want the column name to follow SELECT. Press F11 to see Text column.

    The Select and Sequence Columns display appears, showing the columns existing in the PEOPLE table.

11. Type a 1 in the *Seq* column across from the *NAME* column.

12. Type a 2 in the *Seq* column across from the *SOCSEC* column.

```
                        Select and Sequence Columns

Type sequence numbers (1-999) to select columns, press Enter.

Seq   Column      Table/view   Collection   Text
 1    NAME        PEOPLE       YOURDB2
      EMPLNO      PEOPLE       YOURDB2      Employee ID no.
 2    SOCSEC      PEOPLE       YOURDB2
      COADDR      PEOPLE       YOURDB2      Company mail address
      STRADDR     PEOPLE       YOURDB2      Street address
      CITY        PEOPLE       YOURDB2
      ZIP         PEOPLE       YOURDB2
      COPHONE     PEOPLE       YOURDB2      Employee internal telephone number.
      PERPHONE    PEOPLE       YOURDB2      Employee home telephone number.
      AGE         PEOPLE       YOURDB2
```

13. Press the Enter key.

    The Enter SQL Statements display appears again with the NAME, SOCSEC appearing after SELECT.

```
                         Enter SQL Statements

Type SQL statement, press Enter.
===> SELECT NAME, SOCSEC
     FROM YOURDB2.PEOPLE
```

14. Press the Enter key.

    The statement you created is now syntax checked, validity checked, and run, according to the statement processing options that you previously selected.

    If you ran the statement and if no errors were encountered, the SQL Statement Entry display looks like this:

```
                         Enter SQL Statements

Type SQL statement, press Enter.
     SELECT NAME, SOCSEC
     FROM YOURDB2.PEOPLE
     SELECT statement run complete.
===>
```

After First Use in Session

Once you have used the list function, the values you selected remain in effect until you change them or until you change the list of libraries option on the Work with Session Services display.

Note: To change the lists, remember this rule:

Always select the list you are primarily interested in. For example, if you want a list of columns, but you believe that the columns you want are in a collection or table not currently selected, press F18 (Select columns). Then, from the column list display, press the appropriate function key to change the table or collection. Do not go first to the collection or table list, in this case, because you do not want to have the table name or collection name inserted into your statement.

Testing Your SQL Statements Using Interactive SQL

An important use for interactive SQL is to test your SQL statements before embedding them into your host program. When exiting interactive SQL, save your session in a source file. Then, use the system source entry utility (SEU) to copy the statements into your program.

Entering DBCS Data

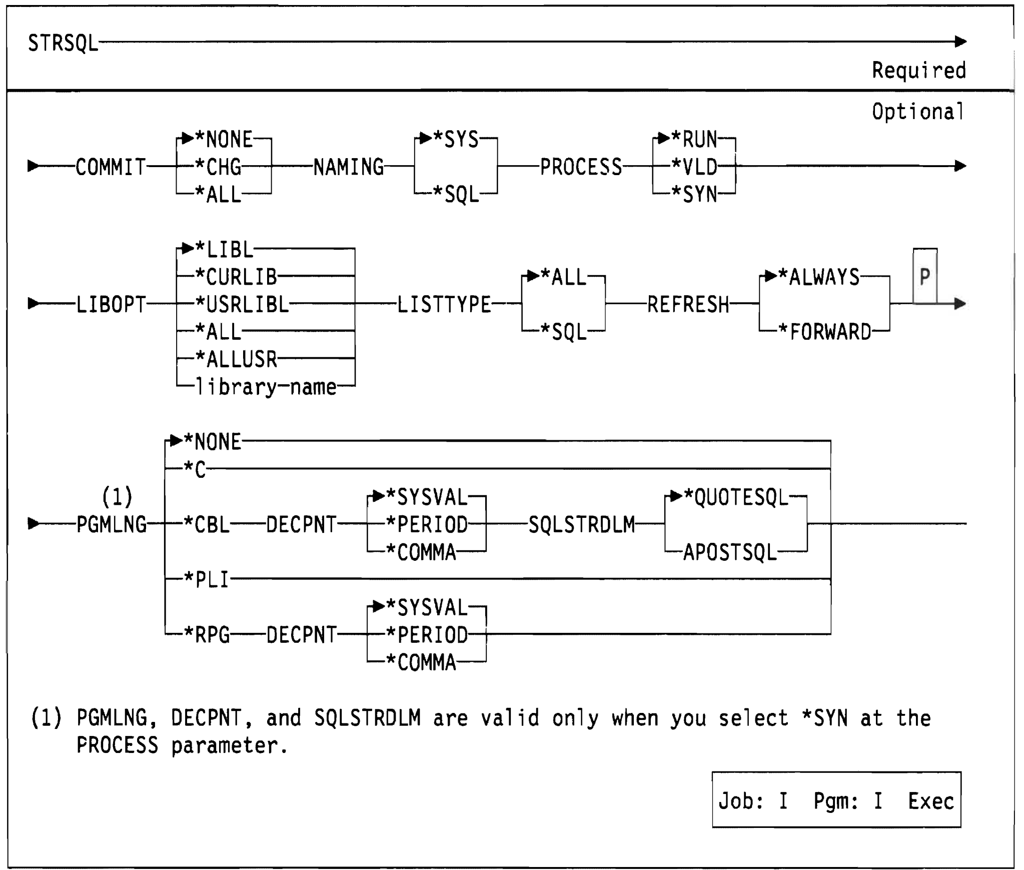
When you enter double-byte character set (DBCS) data on the Enter SQL Statements display, you must be aware of how the shift-out and shift-in characters are processed by interactive SQL. Each line of data must contain as many shift-out characters as shift-in characters. To assist processing a DBCS data string requiring more than one line for entry, interactive SQL removes the extra shift-out and shift-in characters. If the very last (farthest right) column on the line contains a shift-in character and the very first (farthest left) column of the next line contains a shift-out character, these shift-in, shift-out characters are removed by interactive SQL when the statement is processed. Also, when the two farthest right columns of a line contain a shift-in character, followed by a single-byte blank character, and the next line contains a shift-out character in the farthest left column, the shift-in, blank, shift-out sequence is removed when the statement is processed.

STRSQL Command

The STRSQL command starts interactive SQL, which immediately shows the Enter SQL Statements display. This display allows you to build, edit, enter, and run an SQL statement in an interactive environment. Any messages during the running of the program are shown on this display.

You can also specify many processing options by typing STRSQL on the command line and then pressing F4 (prompt). All of the parameters for the STRSQL command, except NAMING, can be set by using the Work with Session Services display in interactive SQL.

A syntax diagram of the command and a definition of the parameters follows:



COMMIT
Specifies whether the SQL statements are run under commitment control.

<u>*NONE:</u> Specifies that commitment control is not used. COMMIT and ROLLBACK statements are not allowed. If SQL DDL statements are run, *NONE must be specified.

***CHG:** Specifies that only those rows updated, deleted, or inserted are locked until the unit of recovery (transaction) is committed or rolled back.

***ALL:** Specifies that all rows selected, updated, deleted, and inserted are locked until the unit of recovery (transaction) is committed or rolled back.

NAMING
Specifies the naming convention used for naming objects in SQL statements.

<u>***SYS:**</u> Indicates that the system naming convention is used (library-name/file-name).

***SQL:** Indicates that the SQL naming convention is used (collection-name.table-name).

PROCESS
Specifies what values are used to process the statements.

<u>***RUN:**</u> Specifies that the statements are syntax checked, data checked, and then run.

***VLD:** Specifies that the statements are syntax checked and data checked, but not run.

***SYN:** Specifies that the statements are syntax checked only.

LIBOPT
Specifies which collections and libraries are used as a basis for building a list when F16, F17, F18, or F4 on prompts is pressed.

<u>***LIBL:**</u> Specifies that all the libraries in the user and system portions of the job's library list are shown.

***CURLIB:** Specifies that the current library is shown.

***USRLIBL:** Specifies that only the libraries in the user portion of the job's library list are shown.

***ALL:** Specifies that all the libraries in the system, including QSYS, are shown.

***ALLUSR:** Specifies that all the nonsystem libraries, which include all user-defined libraries and the QGPL library, that are not in the job's library list are shown. Libraries whose names start with the letter Q, other that QGPL, are not included.

library-name: Allows you to specify the name of a library.

LISTTYPE
Specifies what types of objects are displayed with list support (F16, F17, F18, or F4 on prompts).

<u>***ALL:**</u> Specifies that all objects are displayed.

***SQL:** Specifies that only SQL created objects are displayed.

REFRESH
Specifies when the displayed SELECT output data is refreshed.

<u>***ALWAYS:**</u> Specifies that the data is normally refreshed when scrolling forward and backward.

***FORWARD:** Specifies that the data is refreshed only as the user scrolls forward to the end of the data for the first time. When scrolling backward, a copy of the data already seen is displayed.

PGMLNG
Specifies the program language syntax rules to use. To use this parameter, you must select *SYN at the PROCESS parameter.

<u>***NONE:**</u> Specifies that you are not using a specific language's syntax check rules, but you are using a superset of all of the other languages' syntax rules.

The supported languages are:

***C:** Specifies that you are checking syntax using the C language syntax rules.

***CBL:** Specifies that you are checking syntax using the COBOL language syntax rules.

***PLI:** Specifies that you are checking syntax using the PL/I language syntax rules.

***RPG:** Specifies that you are checking syntax using the RPG language syntax rules.

DECPNT
Specifies what kind of decimal point you want to use if you are using COBOL (*CBL) or RPG (*RPG) language syntax rules.

You are prompted to use one of the following types of decimal points:

***SYSVAL:** Specifies that the decimal point is extracted from the system value.

***PERIOD:** Specifies that a period represents the decimal point.

***COMMA:** Specifies that a comma represents the decimal point.

SQLSTRDLM
Specifies the SQL string delimiter. To use this parameter, you must be using COBOL (*CBL).

***QUOTESQL:** Specifies that a quotation mark represents the SQL string delimiter.

***APOSTSQL:** Specifies that an apostrophe represents the SQL string delimiter.

Example

```
STRSQL PROCESS(*SYN) PGMLNG(*CBL) NAMING(*SQL) DECPNT(*COMMA)
  SQLSTRDLM(*APOSTSQL)
```

This command starts an interactive SQL session that only syntax checks SQL statements. The syntax rules used by the syntax checker is from the COBOL language. The SQL naming convention is used for this session. The decimal point will be represented by a comma and the SQL string delimiter will be represented by an apostrophe.

Chapter 12. SQL Data Protection

This chapter describes the security plan for protecting SQL data from unauthorized users and the methods for ensuring data integrity.

SQL Security

All objects on the AS/400 system, including SQL objects, are managed by the system security function. SQL uses the GRANT and REVOKE statements to interface with the AS/400 system commands Grant Object Authority (GRTOBJAUT) and Revoke Object Authority (RVKOBJAUT). For more information on system security and the use of the GRTOBJAUT and RVKOBJAUT commands, see the *Security Concepts and Planning* manual.

The SQL GRANT and REVOKE statements only operate on tables and views. In some cases, it is necessary to use system GRTOBJAUT and RVKOBJAUT commands to authorize users to other objects, such as commands and programs.

The authority checked for SQL statements depends on whether the statement is static, dynamic, or being run interactively. For static SQL statements, authority is checked against the user who is the owner of the program containing the SQL statement in addition to the user who is running the program. This is accomplished by using USRPRF(*OWNER) when the precompiler calls the compiler.

For dynamic SQL statements or for statements issued interactively, authority is checked against the user who is running the program or is processing the statement.

Authorization ID

The authorization ID identifies a unique user and, in the SQL/400 program, is a user profile object. Authorization IDs can be created using the system Create User Profile (CRTUSRPRF) command.

Public Authority

Public authority on specific tables and views is controlled by using the SQL GRANT and REVOKE statements. If *PUBLIC is specified with the SQL GRANT statement, authority to an object is granted to all users that have no private authority to that object.

For more information on the GRANT and REVOKE statements, see the *SQL/400 Reference* manual.

Views

A view can prevent unauthorized users from having access to sensitive data. The application program can access the data it needs in a table, without having access to sensitive or restricted data in the table. A view can restrict access to particular columns by not specifying those columns in the SELECT list (for example, employee salaries). A view can restrict access to particular rows in a table by specifying a WHERE clause (for example, allowing access only to the rows associated with a particular department number).

© Copyright IBM Corp. 1988, 1989

SQL Data Integrity

SQL data integrity protects data from being destroyed or changed by unauthorized persons, system operation or hardware failures (such as physical damage to a disk), programming errors, interruptions before a job is completed (such as a power failure), or interference from running applications at the same time (such as serialization problems). Data integrity is ensured by the following functions:

- Concurrency
- Atomic operations
- Journaling
- Commitment control
- Save/restore
- Damage tolerance
- Index recovery

For more information about each of these functions, see the *Database Guide* and the *Backup and Recovery Guide*.

Concurrency

Concurrency is the ability for multiple users to access and change data in the same table or view at the same time without risk of losing data integrity. This ability is automatically supplied by the AS/400 database manager. Locks are implicitly acquired on tables and rows to protect concurrent users from changing the same data at precisely the same time.

In some cases, the program may acquire locks that prevent other statements in the same program from running. For example, a lock on a row currently held by one cursor will prevent another cursor in the same program (or in a DELETE or UPDATE statement not associated with the cursor) from acquiring a lock on the same row.

Deadlock detection is not provided; instead, default and user-specifiable lock-wait time-out values are supported. SQL creates tables, views, and indexes with the default record wait time (60 seconds) and the default file wait time (*IMMED). The user may change these values by using the Change Physical File (CHGPF), Change Logical File (CHGLF), and Override Database File (OVRDBF) commands. (For more information on these commands, see the *CL Reference* manual.)

You may explicitly prevent other users from using a table at the same time by using the SQL LOCK TABLE statement, which is described in the *SQL/400 Reference* manual.

Atomic Operations

In general, all underlying database data definition functions are designed to be atomic (either they will complete or they will appear to never have been started). This is true regardless of when or how the function was ended or interrupted (power failure, abnormal ending, job cancel, and so forth). Data definition statements are not affected by the COMMIT and ROLLBACK statements. However, because the underlying database data definition functions are atomic, the database is never left in an unusable state.

The following SQL data definition statements are guaranteed to be atomic:

COMMENT ON	GRANT (See note.)
DROP TABLE	LABEL ON
DROP VIEW	REVOKE (See note.)
DROP INDEX	

Note: If multiple tables are specified for a GRANT or REVOKE statement, the tables are processed one at a time, so the entire SQL statement is not atomic, but the GRANT or REVOKE to each individual table will be atomic.

The following data definition statements are not atomic because they involve more than one OS/400 database operation:

CREATE COLLECTION
CREATE TABLE
CREATE VIEW
CREATE INDEX
DROP COLLECTION

For example, a CREATE TABLE may be interrupted after the AS/400 physical file has been created, but before the member has been added. Therefore, in the case of create statements, if an operation ends abnormally, you may have to drop the object and then create it again. In the case of a DROP COLLECTION statement, you may have to drop the collection again or use the DLTLIB CL command to remove the remaining parts of the collection.

Journaling

The AS/400 journal support supplies an audit trail and forward and backward recovery. Forward recovery can be used to take an older version of a table and apply the changes logged on the journal to the table. Backward recovery can be used to remove changes logged on the journal from the table.

When an SQL collection is created, a journal and journal receiver are created in the collection. The journal and journal receiver are not created on a user auxiliary storage pool (ASP). However, because placing journal receivers on ASPs can improve performance, the user who manages the journal may wish to create all future receivers on an ASP.

When a table is created, it is automatically journaled to the journal SQL created in the collection. After this point, it is the user's responsibility to use the journal functions to manage the journal, the journal receivers, and the journaling of tables to the journal. For example, if a table is moved into a collection, no automatic change to the journaling status occurs. If a table is restored, the normal journal rules apply. That is, if the table was journaled at the time of the save, it is journaled to the same journal at restore time. If the table was not journaled at the time of the save, it is not journaled at restore time.

A user can stop journaling on any table using the journal functions, but doing so prevents SQL from running under commitment control. SQL is still able to function in this case if the user has specified COMMIT(*NONE); however, this does not provide the same level of integrity that journaling and commitment control provide.

Commitment Control

The AS/400 commitment control provides a means to process a group of database changes (UPDATES, INSERTS, or DELETES) as a single unit of recovery (transaction). An SQL COMMIT statement guarantees that the group of operations is completed. An SQL ROLLBACK statement guarantees that the group of operations is backed out.

If the user requests COMMIT (*CHG) and COMMIT (*ALL) when the program was precompiled or when interactive SQL was started, then SQL sets up the commitment control environment by implicitly invoking the Start Commitment Control (STRCMTCTL) command. The specified COMMIT keyword value is used when SQL starts commitment control. NFYOBJ(*NONE) is specified when SQL starts commitment control. The user can issue STRCMTCTL before invoking SQL in order to specify different NFYOBJ or LCKLVL parameters.

If the user specifies commitment control and the application does not complete for any reason or if it requests a ROLLBACK, all updates, inserts, and deletes made within the unit of recovery are backed out, even if a power failure occurs.

The journal created in the SQL collection is normally the journal used for logging all changes to SQL tables. The user may, however, use the system journal functions to journal SQL tables to a different journal. This is necessary if tables from multiple collections need to be used in the same unit of recovery. This is because AS/400 commitment control requires that all files under commitment control are journaled to the same journal.

The AS/400 system uses locks on rows to keep other jobs from accessing changed data before a unit of recovery completes. If COMMIT(*ALL) is specified, locks on rows fetched are also used to prevent other jobs from changing data that was read before a unit of recovery completes. This ensures that, if the same unit of recovery rereads a record, it gets the same result.

Commitment control handles up to 4096 distinct row changes in a unit of recovery. If COMMIT(*ALL) is specified, all rows read are also included in the 4096 limit. (If a row is changed or read more than once in a unit of recovery, it is only counted once toward the 4096 limit.) Holding a large number of locks adversely affects system performance and does not allow concurrent users to access rows locked in the unit of recovery until the end of the unit of recovery. It is, therefore, in the user's best interest to keep the number of rows processed in a unit of recovery small.

The HOLD value on COMMIT and ROLLBACK allows the user to keep the cursor open and start another unit of recovery without issuing an OPEN again.

If there are locked rows (records) pending from running a SQL precompiled program or an interactive SQL session, a COMMIT or ROLLBACK statement can be issued from the system Command Entry display. Otherwise, an implicit ROLLBACK operation occurs when the job is ended.

Commitment control does not apply to data definition statements.

Table 12-1. Record Lock Duration

SQL Statement	COMMIT Parameter	Duration of Record Locks
SELECT INTO	*NONE *CHG *ALL (See note 2)	No locks No locks From read until ROLLBACK or COMMIT
FETCH (read-only cursor)	*NONE *CHG *ALL (See note 2)	No locks No locks From read until ROLLBACK or COMMIT
FETCH (update or delete capable cursor) See note 1	*NONE *CHG *ALL	From read until next FETCH When record not updated or deleted from read until next FETCH When record is updated or deleted from read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT
INSERT	*NONE *CHG *ALL	No locks From insert until ROLLBACK or COMMIT From insert until ROLLBACK or COMMIT
UPDATE (non-cursor)	*NONE *CHG *ALL	Each record locked while being updated From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT
DELETE (non-cursor)	*NONE *CHG *ALL	Each record locked while being deleted From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT
UPDATE (with cursor)	*NONE *CHG *ALL	Lock remains until next FETCH From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT
DELETE (with cursor)	*NONE *CHG *ALL	Lock remains until next FETCH From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT

Notes:

1. A cursor is open with UPDATE or DELETE capabilities if the result table is not read-only (see description of DECLARE CURSOR in *SQL/400 Reference*) and if one of the following is true:
   - The cursor is defined with a FOR UPDATE CLAUSE.
   - The cursor is defined without a FOR UPDATE or ORDER BY clause and the program contains at least one of the following:
     - Cursor UPDATE referencing the same cursor-name
     - Cursor DELETE referencing the same cursor-name
     - An EXECUTE or EXECUTE IMMEDIATE statement
2. In the following cases, a table or view may be locked exclusively in order to satisfy COMMIT(*ALL):
   - When commitment control has already started and is using COMMIT(*CHG) and changes are pending to the database.
   - If a subselect is processed that includes a join, group by, or union, or if the processing of the query requires the use of a temporary result, an exclusive lock is acquired to protect the user from seeing uncommitted changes.

Save/Restore

The AS/400 save/restore functions are used to save tables, views, indexes, journals, journal receivers, and collections on disk (save file) or to some external media (tape or diskette). The saved versions may be restored onto the system at some later time. The save/restore function allows an entire collection, selected objects, or only objects changed since a given date and time to be saved. All information needed to restore an object to its previous state is saved. This function can be used to recover from damage to individual tables by restoring the data with a previous version of the table or the entire collection.

Damage Tolerance

The AS/400 database functions have a certain amount of tolerance to damage caused by disk errors or system errors.

A DROP operation always succeeds, regardless of the damage. This assures that should damage occur, at least the table, view, or index can be deleted and restored or created again.

In the event that a disk error has damaged a small portion of the rows in a table, the AS/400 database manager allows the user to read rows still accessible.

Index Recovery

The AS/400 database manager supplies several functions to deal with index recovery. All indexes on the system have a maintenance option that specifies when an index is maintained. SQL indexes are created with an attribute of *IMMED maintenance.

In the event of a power failure or abnormal system failure, indexes that are in the process of change may need to be rebuilt by the database manager to make sure they agree with the actual data. All indexes on the system have a recovery option that specifies when an index should be rebuilt if necessary. All SQL indexes with an attribute of UNIQUE are created with a recovery attribute of *IPL (this means that these indexes are rebuilt before the OS/400 has been started). All other SQL indexes are created with the *AFTIPL recovery option (this means that after the operating system has been started, indexes are asynchronously rebuilt). During an IPL, the operator can see a display showing indexes needing to be rebuilt and their recovery option. The operator may override the recovery options.

The AS/400 database manager supplies an index journaling function that makes it unnecessary to rebuild an entire index due to a power or system failure. If the index is journaled, the system database support automatically makes sure the index is in synchronization with the data in the tables without having to rebuild it from scratch. SQL indexes are *not* journaled automatically. The user can, however, use the Start Journal Access Path (STRJRNAP) CL command[1] to journal any index created by SQL.

The save/restore function allows the user to save indexes when a table is saved by using ACCPTH (*YES) on the Save Object (SAVOBJ) or Save Library (SAVLIB) CL commands. In the event of a restore when the indexes have also been saved, there

1 For more information on control language commands, see the *CL Reference* manual.

is no need to rebuild the indexes. Any indexes not previously saved and restored are automatically and asynchronously rebuilt by the database manager.

Catalog Integrity

Catalogs contain information about tables, views, and indexes in a collection. The database manager ensures that the information in the catalog is accurate at all times. This is accomplished by preventing end users from explicitly changing any information in the catalog and by implicitly maintaining the information in the catalog when changes occur to the tables, views, and indexes described in the catalog.

The integrity of the catalog is maintained whether objects in the collection are changed by SQL statements, OS/400 CL commands, System/38 Environment CL commands, System/36 Environment functions, or any other product or utility on an AS/400 system. For example, deleting a table can be done by running an SQL DROP statement, issuing an OS/400 DLTF CL command, issuing a System/38 DLTF CL command or entering option 4 on a WRKF or WRKOBJ display. Regardless of the interface used to delete the table, the database manager will remove the description of the table from the catalog at the time the delete is performed. The following is a list of functions and the associated effect on the catalog:

Table 12-2. Effect of Various Functions on Catalogs

Function	Effect on the Catalog
Create object into collection	Information added to catalog
Delete of object from collection	Related information removed from catalog
Restore of object into collection	Information added to catalog
Change of object long comment	Comment updated in catalog
Change of object label (text)	Label updated in catalog
Change of object owner	Owner updated in catalog
Move of object from a collection	Related information removed from catalog
Move of object into collection	Information added to catalog
Rename of object	Name of object updated in catalog

Chapter 13. Testing SQL Statements in Application Programs

This chapter describes how to establish a test environment for SQL statements in an application program and how to debug this program.

Establishing a Test Environment

Some things you need to test your program are:

- **Authorization.** You need to be authorized to create tables and views, access SQL data, and create and run programs.
- **A test data structure.** If your program updates, inserts, or deletes data from tables and views, *you should use test data* to verify the running of the program. If your program only retrieves data from tables and views, you might consider using production-level data when testing your program. It is recommended, however, that you use the Start Debug (STRDBG) command with UPDPROD(*NO) to assure that the production level data does not accidentally get changed. See the chapter on testing in the *CL Programmer's Guide* for more information on debugging.
- **Test input data.** The input data your program uses during testing should be valid data that represents as many possible input conditions as you can think of. You cannot be sure that your output data is valid unless you use valid input data.

  If your program verifies that input data is valid, include both valid and invalid data to verify that the valid data is processed and the invalid data is detected.

  You might have to refresh the data for subsequent tests.

To test the program thoroughly, test as many of the paths through the program as possible. For example:

- Use input data that forces the program to run each of its branches.
- Check the results. For example, if the program updates a row, select the row to see if it was updated correctly.
- Be sure to test the program error routines. Again, use input data that forces the program to encounter as many of the anticipated error conditions as possible.
- Test the editing and validation routines your program uses. Give the program as many different combinations of input data as possible to verify that it correctly edits or validates that data.

Designing a Test Data Structure

To test an application that accesses SQL data, you may have to create test tables and views:

- **Test views of existing tables.** If your application does not change data and the data exists in one or more production-level tables, you might consider using a view of the existing tables. It is also recommended that you use STRDBG command with UPDPROD(*NO) to assure that the production level data does not accidentally get changed. See the chapter on testing in the *CL Programmer's Guide* for more information on debugging.

© Copyright IBM Corp. 1988, 1989

- **Test tables.** When your application creates, changes, or deletes data, you will probably want to test the application by using tables that contain test data. See Chapter 2 for a description of how to create tables and views.

  Also, you may want to use the CRTDUPOBJ CL command to create a duplicate test table, view, or index. See the *CL Reference* manual for more information on using the CRTDUPOBJ command.

Authorization

Before you can create a table, you must be authorized to create tables and to use the collection in which the table is to reside. In addition, you must have authority to create and run the programs you want to test.

If you intend to use existing tables and views (either directly or as the basis for a view), you must be authorized to access those tables and views.

If you want to create a view, you must be authorized to create views and must have authorization to each table and view on which the view is based. For more information on specific authorities required for any specific SQL statement, see the *SQL/400 Reference* manual.

Debugging Your Program

Debugging your program with SQL statements is much the same as debugging your program without SQL statements. However, when running your program with SQL statements in the STRDBG environment, SQL puts a message in the job log about how the SQL statements ran. This message is an indication of the SQLCODE for the SQL statement. If the statement ran successfully, the SQLCODE value is zero, and a completion message is issued. A negative SQLCODE results in a diagnostic message. A positive SQLCODE results in an informational message.

The message is a 4-digit code prefixed by SQL. For example, an SQLCODE of −204 results in a message of SQL0204.

References to high-level language statement numbers in debug must be taken from the compile listing.

Chapter 14. Guidelines and Techniques for Using SQL

This chapter describes the guidelines and techniques for using SQL statements in application programs. The chapter consists of two sections. The first section, "Guidelines for Using SQL Statements," describes guidelines for designing a program that uses SQL and system resources more efficiently. The second section, "Techniques for Solving Some Common Collection Problems" on page 14-8, suggests techniques for using SQL statements in an application program and for solving some common database problems.

Guidelines for Using SQL Statements

This section describes some specific considerations and guidelines to help you tune the SQL statements in an application program. As a general rule, you can ignore most of these guidelines and still get correct results when accessing SQL data. These guidelines help you design a program that minimizes its use of SQL and system resources and that minimizes the time needed to access SQL data from a very large table.

The SQL language is a high-level language with much flexibility. Because of this, you can sometimes write a select-statement several different ways to retrieve the same data. However, the performance of different forms of a select-statement can vary greatly. In this section are several examples of alternative SQL statements. The recommendations given with each example are based on the relative performance of the example.

Some of the suggestions are easy to carry out without compromising any of the ease-of-use functions of SQL. Others, however, are complex and difficult to use. To determine if you need to make the effort to tune the performance of your SQL statements, consider the following guidelines:

- If you are accessing a table of 10,000 rows or more, you should start thinking about the performance implications of your SQL statements.
- If you are accessing a table of 100,000 rows or more, you should seriously consider the performance implications of your SQL statements.
- If the SQL statement you issue requires the ordering of 1000 rows or more, you should consider trying to improve the performance of the ORDER BY.
- If you are accessing more than one table (for example, a join), you should consider trying to improve the select-statement performance.

You can improve performance by effectively using an SQL index and by effectively selecting data from two or more tables, described in the following paragraphs.

© Copyright IBM Corp. 1988, 1989

Effectively Using an SQL Index

SQL provides two basic means for accessing tables: a table scan (sequential) and an index-based (direct) retrieval. Index-based retrieval is usually more efficient than table scan. However, when a very large percentage of pages are retrieved, table scan is more efficient than index-based retrieval.

If SQL cannot use an index to access the data in a table, it will have to read all the data in the table. Very large tables present a special performance problem: the high cost of retrieving all the data in the table. The following suggestions help you to design code that allows SQL to take advantage of available indexes.

1. **Avoid numeric conversions.**

   When a column value and a host variable (or literal value) are being compared, try to specify the same data types and attributes. SQL does not use an index for the named column if the host variable or literal value has a greater precision than the precision of the column. If the two items being compared have different data types, SQL will have to convert one or the other of the values, which may result in inaccuracies (because of limited machine precision). For example, EDUCLVL is a halfword integer value (SMALLINT). Specify:

   ```
   ... WHERE EDUCLVL < 11 AND
             EDUCLVL >= 2
   ```

   instead of

   ```
   ... WHERE EDUCLVL < 1.1E1 AND
             EDUCLVL > 1.3
   ```

2. **Avoid character string padding.**

   Try to use the same data length when comparing a fixed-length character string column value to a host variable or literal value. SQL does not use an index if the literal value or host variable is longer than the column length. For example, EMPNO is CHAR(6) and DEPTNO is CHAR(3). Specify:

   ```
   ... WHERE EMPNO > '000300' AND
             DEPTNO < 'E20'
   ```

   instead of

   ```
   ... WHERE EMPNO > '000300 ' AND
             DEPTNO < 'E20 '
   ```

3. **Avoid the use of LIKE patterns beginning with % or _.**

   The percent sign (%), and the underscore (_), when used in the pattern of a LIKE predicate, specify a character string that is similar to the column value of rows you want to select. When used to denote characters in the middle or at the end of a character string, as in

   ```
   ... WHERE LASTNAME LIKE 'J%SON%'
   ```

   they can take advantage of SQL indexes. However, when used at the beginning of a character string, as in

   ```
   ... WHERE LASTNAME LIKE '%SON'
   ```

   they may prevent SQL from using any indexes that might be defined on the *LASTNAME* column to limit the number of rows scanned. You should therefore avoid using these symbols at the beginning of character strings, especially if you are accessing a particularly large table.

4. **Be aware that SQL does not use an index in the following instances:**

   - For a column that is expected to be updated; for example, your program might include

     ```
     EXEC SQL
       DECLARE DEPTEMP CURSOR FOR
         SELECT EMPNO, LASTNAME, DEPTNO
           FROM USER1.TEMPL
           WHERE (DEPTNO = 'D11' OR
                  DEPTNO = 'D21') AND
                 EMPNO >= '000190'
           FOR UPDATE OF EMPNO, DEPTNO
     END-EXEC.
     ```

     even if you do not intend to update the employee's serial number. In this example, SQL cannot use an index with a key of EMPNO or DEPTNO.

     SQL can operate more efficiently if the FOR UPDATE OF column list only names the column you intend to update: *DEPTNO*. Therefore, do not specify a column in the FOR UPDATE OF column list unless you intend to update the column.

   - For a column being compared with another column from the same row. For example:

     ```
     EXEC SQL
       DECLARE DEPTDATA CURSOR FOR
         SELECT DEPTNO, DEPTNAME
           FROM USER1.TDEPT
           WHERE DEPTNO = ADMRDEPT
     END-EXEC.
     ```

     Even though there is an index for *DEPTNO* and another index for *ADMRDEPT*, SQL will not use either index. The index has no added benefit because every row of the table needs to be looked at.

5. **An application program can do retrievals based on some key value in either of two ways:**

   a. A SELECT INTO statement can be used with a host variable, or

   b. A cursor could be declared by using a host variable, and a sequence of OPEN, FETCH, and CLOSE statements can be issued.

For example, suppose you want to FETCH a row from a table based on some value input from a display station:

```
EXEC SQL
  DECLARE EMPCSR CURSOR FOR
    SELECT EMPNO, LASTNAME, DEPTNO
      FROM USER1.TEMPL
      WHERE EMPNO = :EMPVAR
END-EXEC.

(Input an employee number from the display station.)

EXEC SQL
  OPEN EMPCSR
END-EXEC.

EXEC SQL
  FETCH EMPCSR
    INTO :EMPVAR, :LASTVAR, :DEPTVAR
END-EXEC.

EXEC SQL
  CLOSE EMPCSR
END-EXEC.

(Get the next employee number from the display station
 and repeat the OPEN, FETCH, and CLOSE.)
```

If the WHERE condition can be completely satisfied by doing key retrievals using an existing index, SQL will not actually close the cursor when the CLOSE statement is issued. The OPEN statement must still be issued, before the next FETCH, but since the full open is not done, the application runs much faster. The cursor is completely closed when the program ends.

Instead, the following SELECT INTO statement could be specified:

```
(Input an employee number from the display station.)

EXEC SQL
  SELECT EMPNO, LASTNAME, DEPTNO
    INTO :EMPVAR, :LASTVAR, :DEPTVAR
    FROM USER1.TEMPL
    WHERE EMPNO = :EMPVAR
END-EXEC.

(Get the next employee number from the display station
 and repeat the SELECT.)
```

If the WHERE condition can be completely satisfied by doing key retrievals using an existing index, SQL will leave open the internal cursor used to select the records until the program ends. If the SELECT INTO statement is run again, a full open is not necessary and the application runs much faster.

Improving Performance When Selecting Data from Two or More Tables

If the select-statement you are considering accesses two or more tables, all the recommendations suggested in the previous section apply. The following suggestion is directed specifically to select-statements that access several tables.

You might want to provide redundant information when joining several tables. If you give SQL extra information to work with when requesting a join, it can better determine the best way to do the join. The additional information might seem redundant, but it is helpful to SQL. For example, instead of coding:

```
EXEC SQL
  DECLARE DEPTDATA CURSOR FOR
    SELECT USER1.TEMPL.DEPTNO, DEPTNAME, EMPNO, LASTNAME
      FROM USER1.TEMPL, USER1.TPROJ
      WHERE USER1.TEMPL.DEPTNO = USER1.TPROJ.DEPTNO AND
            USER1.TPROJ.DEPTNO = :DEPTNUM
END-EXEC.
 ...
```

provide SQL with a little more data in the WHERE clause:

```
EXEC SQL
  DECLARE DEPTDATA CURSOR FOR
    SELECT USER1.TEMPL.DEPTNO, DEPTNAME, EMPNO, LASTNAME
      FROM USER1.TEMPL, USER1.TPROJ
      WHERE USER1.TEMPL.DEPTNO = USER1.TPROJ.DEPTNO AND
            USER1.TPROJ.DEPTNO = :DEPTNUM AND
            USER1.TEMPL.DEPTNO = :DEPTNUM
END-EXEC.
 ...
```

SQL may be able to make a more informed decision about the best way to process the select-statement when given this redundant information.

For joins involving more than two tables, providing redundant information might become tedious. To improve performance and minimize the amount of redundant information, code a predicate that refers to an indexed column.

In the above example, assume that *DEPTNO* column of USER1.TEMPL is indexed but *DEPTNO* column of USER1.TPROJ is not. To improve the statement performance (without providing redundant information), you can code:

```
EXEC SQL
  DECLARE DEPTDATA CURSOR FOR
    SELECT USER1.TEMPL.DEPTNO, DEPTNAME, EMPNO, LASTNAME
      FROM USER1.TEMPL, USER1.TPROJ
      WHERE USER1.TEMPL.DEPTNO = USER1.TPROJ.DEPTNO AND
            USER1.TEMPL.DEPTNO = :DEPTNUM
END-EXEC.
 ...
```

Improving Performance by Reducing the Number of Opens

The number of opens in a program can significantly affect performance. An open occurs on:

- An OPEN statement
- SELECT INTO statement
- An INSERT statement with a VALUES clause
- An UPDATE statement with a WHERE condition
- An UPDATE statement with a WHERE CURRENT OF cursor and SET clauses that refer to operators or functions
- A DELETE statement with a WHERE condition

An INSERT statement with a subselect requires two opens.

To minimize the number of opens, the SQL/400 program leaves a cursor open in the following cases and reuses the cursor if the statement is run again:

- An INSERT statement with a VALUES list
- A SELECT INTO statement when the WHERE clause can be completely satisfied by doing key retrievals from an existing index and when GROUP BY, HAVING, or column functions have not been specified
- OPEN statement when the WHERE clause in the associated DECLARE CURSOR can be completely satisfied by doing key retrievals from an existing index and when GROUP BY, HAVING, or column functions have not been specified

The SQL/400 program only reuses cursors opened by the same statement number. An identical statement coded later in the program does not reuse a cursor from any other statement. If the identical statement must be run in the program many times, code it once in a procedure and call the procedure to run the function.

The SQL/400 program does an open for each execution of an UPDATE WHERE CURRENT OF when any expression in the SET clause contains an operator or function. The open can be avoided by coding the function or operation in the host language code.

For example, the following UPDATE will cause the SQL/400 program to do an open for every execution:

```
EXEC SQL
 FETCH EMPT  INTO  :SALARY
END-EXEC

EXEC SQL
 UPDATE USER1.TEMPL
    SET SALARY = :SALARY + 1000
 WHERE CURRENT OF EMPT
END-EXEC.
```

Instead, use the following coding technique to avoid opens:

```
EXEC SQL
 FETCH EMPT  INTO  :SALARY
END EXEC.

ADD 1000 TO SALARY.

EXEC SQL
 UPDATE USER1.TEMPL
    SET SALARY = :SALARY
 WHERE CURRENT OF EMPT
END-EXEC.
```

The TRCJOB or DSPJRN CL commands can be used to determine the number of opens being performed by an SQL statement. See the *CL Reference* for information on the TRCJOB and DSPJRN commands.

Improving Performance by Using Blocking Considerations

SQL attempts to retrieve and insert records a block at a time when possible to improve performance. The user can control blocking, if desired, by using the SEQONLY parameter on the OVRDBF CL command prior to calling the application program that contains the SQL statements. For more information on the OVRDBF command, see the *CL Reference*.

SQL automatically blocks records in the following cases:

- INSERT

  If an INSERT statement contains a subselect, inserted records are blocked and not actually inserted into the target table until the block is full.

  If an INSERT statement contains a subselect, and COMMIT(*ALL) has not been specified, records for the subselect are retrieved in a block.

  Note: If an INSERT with a VALUES clause is specified, SQL will not actually close the internal cursor used to perform the inserts until the program ends. If the same INSERT statement is run again, a full open is not necessary and the application runs much faster.

- OPEN

  If COMMIT(*NONE) has been specified, and the cursor is only used for FETCH statements, blocking is done when the records are retrieved. Notice that if dynamic SQL statements exist in the program, SQL is unable to determine whether UPDATE or DELETE statements refer to the cursor, so blocking is not done.

Improving Performance when Paging Interactively Displayed Data

In large tables, paging performance is usually degraded because of the REFRESH(*ALWAYS) parameter on the STRSQL command which dynamically retrieves the latest data directly from the table. Paging performance can be improved by specifying REFRESH(*FORWARD).

When interactively displaying data using REFRESH(*FORWARD), the results of a select-statement are copied to a temporary file as you page forward through the display. Other users sharing the table can make changes to the rows while you are displaying the select-statement results. If you page backward or forward to rows

that have already been displayed, the rows shown are those in the temporary file instead of those in the updated table.

The refresh option can be changed on the Session Services display.

Techniques for Solving Some Common Collection Problems

This section provides techniques to help you do the following tasks:

- Page through retrieved data
- Keep a copy of the data
- Retrieve the data a second time
- Establish position at the end of a table
- Add data to the end of a table
- Update data as it is retrieved from the collection
- Update data previously retrieved
- Change the table definition

Paging through Retrieved Data

When a program retrieves data from the database, the FETCH statement allows it to page forward through the data. SQL has no statement equivalent to a backward FETCH. That leaves you with two programming options:

1. Keep a copy of the data that has been fetched and page through it by some programming technique.
2. Use SQL to retrieve the data again, typically by a second cursor.

Both options are discussed in more detail in the following sections.

Keeping a Copy of the Data

One effect of this approach is that by paging backward you always see exactly the same data that was fetched, even if the data in the table has changed in the meantime. That may be an advantage if you need to see a consistent set of data. However, it has the disadvantage of not allowing you to see updates made by others as soon as they are committed to the table.

The locks you hold on the data may prevent others from updating it, which you may or may not want. If COMMIT(*ALL) is specified, the locks are not released after the data is fetched. Therefore, if you want others to be able to update the data you have fetched, and you do not need to see those updates, commit your work after fetching the data.

Retrieving Data a Second Time

To retrieve the data a second time, the technique depends on the order in which you want to see the data again: either from the beginning or from the middle of the result table.

Retrieving from the Beginning

To retrieve the data again from the beginning, merely close the active cursor and reopen it. That positions the cursor at the beginning of the result table. But, unless the program holds an exclusive lock on the table locks on all the data, others may have changed it, and what was the first row of the result table is no longer the first row.

Retrieving from the Middle

To retrieve data a second time from somewhere in the middle of the result table, run a second SELECT statement and declare a second cursor on it. For example:

```
EXEC SQL
  DECLARE A CURSOR FOR
    SELECT * FROM DEPARTMENT
      WHERE LOCATION = 'MINNESOTA'
      ORDER BY DEPTNO
END-EXEC.
```

Suppose that you now want to return to the rows that start with DEPTNO = 'M95', and fetch sequentially from that point. Declare a second cursor:

```
EXEC SQL
  DECLARE B CURSOR FOR
    SELECT * FROM DEPARTMENT
      WHERE LOCATION = 'MINNESOTA'
      AND DEPTNO >= 'M95'
      ORDER BY DEPTNO
END-EXEC.
```

That statement positions the cursor where you want it.

Again, unless the table was locked, rows may have been inserted or deleted by other users. If so, the row with DEPTNO = 'M95' may no longer exist. Or there may now be 20 rows with DEPTNO between M95 and M99, where before there were only 16.

The Order of Rows in the Second Result Table

The rows of the second result table may not appear in the same order. SQL does not consider the order of rows as significant unless the select-statement uses ORDER BY. Therefore, if there are several rows with the same DEPTNO value, the second select-statement may retrieve them in a different order from the first. The rows are guaranteed to be in order by department number only if ORDER BY DEPTNO is specified.

The difference in ordering could occur even if you were to run the same SQL statement, with the same host variables, a second time. For example, indexes could be created or dropped that could affect the access plan.

The ordering is more likely to change if the second select-statement has a predicate that the first did not. SQL may choose to use an index on the new predicate. For example, SQL may choose an index on LOCATION for the first statement in our example, and an index on DEPTNO for the second. Because rows are fetched in order by the index key, the second order need not be the same as the first.

Again, executing PREPARE for two similar select-statements can produce a different ordering of rows even if no indexes are created or dropped. In the example, if there are many different values of LOCATION, SQL could choose an index on LOCATION for both statements. Yet, changing the value of DEPTNO in the second statement

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'MINNESOTA'
  AND DEPTNO >='Z98'
  ORDER BY DEPTNO
```

could cause SQL to choose an index on DEPTNO. Because of the subtle relationships between the form of an SQL statement and the values in it, never assume that two different SQL statements return rows in the same order, unless the order is uniquely determined by an ORDER BY clause.

Retrieving in Reverse Order

If there is only one row for each value of DEPTNO, then the following statement specifies a unique ordering of rows:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'MINNESOTA'
  ORDER BY DEPTNO
```

To retrieve the same rows in reverse order, simply specify that the order is descending, as in this statement:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'MINNESOTA'
  ORDER BY DEPTNO DESC
```

A cursor on the second statement would retrieve rows in exactly the opposite order from a cursor on the first statement. But that is guaranteed *only* if the first statement specifies a *unique* ordering.

For retrieving rows in reverse order, it might be useful to have two indexes on the *DEPTNO* column, one in ascending order and one in descending order.

Establishing Position at the End of a Table

The end of the table is defined as a result of a select-statement like the one that follows:

```
SELECT * FROM DEPARTMENT
  ORDER BY DEPTNO DESC
```

Adding Data to the End of a Table

The order in which rows are returned to your program depends on the ORDER BY clause in the SQL statement. To get the effect of adding data to the end of a table, include a sequence number column in the table definition. Then, when you retrieve data from the table, use an ORDER BY clause naming that column.

Updating Data as It Is Retrieved from a Table

You can update rows of data as you retrieve them. On the select-statement, use FOR UPDATE OF, followed by a list of columns that may be updated. Then use the cursor-controlled UPDATE statement. The WHERE CURRENT OF clause names the cursor that points to the row you want to update. If neither a FOR UPDATE OF nor an ORDER BY clause is specified, all columns may be updated.

Restrictions

If the FOR UPDATE OF clause is specified, you cannot use FOR UPDATE OF with a select-statement that includes any of these elements:

- A column function (AVG, MIN, MAX, SUM, or COUNT)
- The keywords GROUP BY, UNION, or DISTINCT
- A column that is defined with an expression (for example, SALARY + COMMISSION)
- More than one table, as in a join

If a FOR UPDATE OF clause is specified, you cannot update columns that were not named in the FOR UPDATE OF clause. But you may name columns in the FOR UPDATE OF clause that are not in the SELECT list, as in this example:

```
SELECT A, B, C FROM TABLE
  FOR UPDATE OF A,E
```

Do not name more columns than you need in the FOR UPDATE OF clause; indexes on those columns are not used when you access the table.

Updating Data Previously Retrieved

You can page backward and update data that had been previously retrieved by doing one of two things:

1. If you have a second cursor on the data to be updated, and if the select-statement uses none of the above restricted elements, you can use a cursor-controlled UPDATE statement. Name the second cursor in the WHERE CURRENT OF clause.

   Note: This technique does not work for COMMIT (*ALL).

2. In other cases, use UPDATE with a WHERE clause that names all the values in the row or specifies a unique key of the table. You can code one statement, using host variables in the WHERE clause, and run the same statement many times with different values of the variables.

Changing the Table Definition

It is not possible to add, rearrange, or delete columns in a table without dropping and creating the entire table. However, you can dynamically create a view of the table, which includes only the columns you want, in the order you want.

Appendix A. SQL Sample Tables

This appendix contains the sample tables referred to and used in the body of this manual. Along with the tables are the SQL statements for creating the tables and for inserting information into the tables. For detailed information on creating tables, see "Creating and Using a Table" on page 2-1. The tables are:

- Department table (TDEPT)
- Employee table (TEMPL)
- Project table (TPROJ)
- Employee project account table (TEMPRACT)

Creating the Tables

The SQL statements for creating the sample tables are shown in this section.

Department Table (TDEPT)

The sample department table describes each department in the company and specifies the department manager and the next higher department of authority.

This table can be created with the following interactive SQL statement:

```
CREATE TABLE USER1.TDEPT
    (DEPTNO     CHAR(3)     NOT NULL WITH DEFAULT,
     DEPTNAME   CHAR(36)    NOT NULL WITH DEFAULT,
     MGRNO      CHAR(6)     NOT NULL WITH DEFAULT,
     ADMRDEPT   CHAR(3)     NOT NULL WITH DEFAULT)
```

© Copyright IBM Corp. 1988, 1989

Employee Table (TEMPL)

The sample employee table describes the employees of the company.

This table can be created with the following interactive SQL statement:

```
CREATE TABLE USER1.TEMPL
    (EMPNO     CHAR(6)        NOT NULL WITH DEFAULT,
     FIRSTNME  CHAR(12)       NOT NULL WITH DEFAULT,
     MIDINIT   CHAR(1)        NOT NULL WITH DEFAULT,
     LASTNAME  CHAR(15)       NOT NULL WITH DEFAULT,
     DEPTNO    CHAR(3)        NOT NULL WITH DEFAULT,
     PHONENO   CHAR(4)        NOT NULL WITH DEFAULT,
     HIREDATE  CHAR(6)        NOT NULL WITH DEFAULT,
     JOBCODE   DECIMAL(3)     NOT NULL WITH DEFAULT,
     EDUCLVL   SMALLINT       NOT NULL WITH DEFAULT,
     SEX       CHAR(1)        NOT NULL WITH DEFAULT,
     BRTHDATE  CHAR(6)        NOT NULL WITH DEFAULT,
     SALARY    DECIMAL(8,2)   NOT NULL WITH DEFAULT)
```

Project Table (TPROJ)

The sample project table describes each programming project currently active in the company and specifies the department responsible for the project, the department manager, staffing information, and dates marking the duration of the project.

This table can be created with the following interactive SQL statement:

```
CREATE TABLE USER1.TPROJ
    (PROJNO     CHAR(6)       NOT NULL WITH DEFAULT,
     PRNAME     CHAR(36)      NOT NULL WITH DEFAULT,
     DEPTNO     CHAR(3)       NOT NULL WITH DEFAULT,
     DEPTMGR    CHAR(6)       NOT NULL WITH DEFAULT,
     PRSTAFF    DECIMAL(5,2)  NOT NULL WITH DEFAULT,
     PRSTDATE   CHAR(6)       NOT NULL WITH DEFAULT,
     PRENDATE   CHAR(6)       NOT NULL WITH DEFAULT,
     MAJPROJ    CHAR(6)       NOT NULL WITH DEFAULT)
```

Employee Project Account Table (TEMPRACT)

The sample employee project account table describes each programming project currently active in the company and identifies each employee involved in the project, the project number, the account number, the start and end dates of the employee's involvement, and the estimated number of hours the employee will charge to the project.

This table can be created with the following interactive SQL statement:

```
CREATE TABLE USER1.TEMPRACT
    (EMPNO      CHAR(6)       NOT NULL WITH DEFAULT,
     PROJNO     CHAR(6)       NOT NULL WITH DEFAULT,
     ACTNO      SMALLINT      NOT NULL WITH DEFAULT,
     STARTDATE  CHAR(6)       NOT NULL WITH DEFAULT,
     ENDDATE    CHAR(6)       NOT NULL WITH DEFAULT,
     EMPTIME    DECIMAL(5,2)  NOT NULL WITH DEFAULT)
```

Inserting Information into the Tables

Information (data values) is inserted into the tables by using the SQL INSERT statement (see "The INSERT Statement" on page 3-1).

TDEPT Table

The INSERT statement for inserting information into the department table is, in part, as follows:

```
INSERT INTO USER1.TDEPT
       (DEPTNO,
        DEPTNAME,
        MGRNO,
        ADMRDEPT)
  VALUES
       ('A00',
        'COMPUTER SERVICE DIV.',
        '000010',
        '      ')

...and so on for the rest of the information for the table.
```

TEMPL Table

The INSERT statement for inserting information into the employee table is, in part, as follows:

```
INSERT INTO USER1.TEMPL
       (EMPNO,
        FIRSTNME,
        MIDINIT,
        LASTNAME,
        DEPTNO,
        PHONENO,
        HIREDATE,
        JOBCODE,
        EDUCLVL,
        SEX,
        BRTHDATE,
        SALARY)
  VALUES
       ('000010',
        'CHRISTINE',
        'I',
        'HAAS',
        'A00',
        '3978',
        '750101',
         66,
         18,
        'F',
        '330814',
         52750)

...and so on for the rest of the information for the table.
```

TPROJ Table

The INSERT statement for inserting information into the project table is, in part, as follows:

```
INSERT INTO USER1.TPROJ
       (PROJNO,
        PRNAME,
        DEPTNO,
        DEPTMGR,
        PRSTAFF,
        PRSTDATE,
        PRENDATE,
        MAJPROJ)
  VALUES
       ('AD3100',
        'ADMINISTRATION SERVICES',
        'D01',
        '000010',
         6.5,
        '860101',
        '830201',
        '      ')
```

...and so on for the rest of the information for the table.

TEMPRACT Table

The INSERT statement for inserting information into the employee project account table is, in part, as follows:

```
INSERT INTO USER1.TEMPRACT
       (EMPNO,
        PROJNO,
        ACTNO,
        STARTDATE,
        ENDDATE,
        EMPTIME)
  VALUES
       ('000160',
        'MA2100',
         20,
        '860501',
        '860829',
         500)
```

...and so on for the rest of the information for the table.

Sample Tables

The following tables are the results of running the previous CREATE TABLE and INSERT statements:

Department Table (TDEPT)

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
A00	COMPUTER SERVICE DIV.	000010	(blanks)
B01	PLANNING	000020	A00
C01	INFORMATION CENTER	000030	A00
D01	DEVELOPMENT CENTER	(blanks)	A00
E01	SUPPORT SERVICES	000050	A00
D11	MANUFACTURING SYSTEMS	000060	D01
D21	ADMINISTRATION SYSTEMS	000070	D01
E11	OPERATIONS	000090	E01
E21	SOFTWARE SUPPORT	000100	E01

Employee Table (TEMPL)

EMPNO	FIRSTNME	MIDINIT	LASTNAME	DEPTNO	PHONENO
000010	CHRISTINE	I	HAAS	A00	3978 ...
000020	MICHAEL	L	THOMPSON	B01	3476 ...
000030	SALLY	A	KWAN	C01	4738 ...
000050	JOHN	B	GEYER	E01	6789 ...
000060	IRVING	F	STERN	D11	6423 ...
000070	EVA	D	PULASKI	D21	7831 ...
000090	EILEEN	W	HENDERSON	E11	5498 ...
000100	THEODORE	Q	SPENSER	E21	0972 ...
000110	VICENZO	G	LUCCHESI	A00	3490 ...
000120	SEAN	(blanks)	O'CONNELL	A00	2167 ...
000130	DELORES	M	QUINTANA	C01	4578 ...
000140	HEATHER	A	NICHOLLS	C01	1793 ...
000150	BRUCE	(blanks)	ADAMSON	D11	4510 ...
000160	ELIZABETH	R	PIANKA	D11	3782 ...
000170	MASATOSHI	J	YOSHIMURA	D11	2890 ...
000180	MARILYN	S	SCOUTTEN	D11	1682 ...
000190	JAMES	H	WALKER	D11	2986 ...
000200	DAVID	(blanks)	BROWN	D11	4501 ...
000210	WILLIAM	T	JONES	D11	0942 ...
000220	JENNIFER	K	LUTZ	D11	0672 ...
000230	JAMES	J	JEFFERSON	D21	2094 ...
000240	SALVATORE	M	MARINO	D21	3780 ...
000250	DANIEL	S	SMITH	D21	0961 ...
000260	SYBIL	P	JOHNSON	D21	8953 ...
000270	MARIA	L	PEREZ	D21	9001 ...
000280	ETHEL	R	SCHNEIDER	E11	8997 ...

(Continued additional columns for each row are shown on the next page. The *EMPNO* column is repeated for reference only.)

Employee Table (TEMPL) (continued)

EMPNO	HIREDATE	JOBCODE	EDUCLVL	SEX	BRTHDATE	SALARY
000010 ...	750101	66	18	F	330814	52750
000020 ...	731010	61	18	M	480202	41250
000030 ...	750405	60	20	F	410511	38250
000050 ...	690817	58	16	M	450915	40175
000060 ...	730914	55	16	M	450707	32250
000070 ...	800930	56	16	F	530526	36170
000090 ...	700815	55	16	F	410515	29750
000100 ...	800619	54	14	M	561218	26150
000110 ...	680516	58	19	M	491105	46500
000120 ...	731205	58	14	M	421018	29250
000130 ...	710728	55	16	F	350915	23800
000140 ...	761215	55	18	F	460119	28420
000150 ...	720212	55	16	M	470517	25280
000160 ...	771011	54	17	F	550412	22250
000170 ...	780915	54	16	M	510105	24680
000180 ...	730707	53	17	F	490221	21340
000190 ...	740726	53	16	M	520625	20450
000200 ...	760303	55	16	M	410529	27740
000210 ...	790411	52	17	M	530223	18270
000220 ...	780829	55	18	F	480319	29840
000230 ...	761121	53	14	M	350530	22180
000240 ...	791205	55	17	M	540331	28760
000250 ...	791030	52	15	M	391112	19180
000260 ...	750911	52	16	F	361005	17250
000270 ...	800930	55	15	F	530526	27380
000280 ...	770324	54	17	F	360328	26250

Project Table (TPROJ)

PROJNO	PRNAME	DEPTNO	DEPTMGR	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
AD3100	ADMINISTRATION SERVICES	D01	000010	6.5	860101	830201	(blanks)
AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	871001	880215	AD3100
AD3111	PAYROLL PROGRAMMING	D21	000230	2	880101	880401	AD3110
AD3112	PERSONNEL PROGRAMMING	D21	000250	1	870320	870601	AD3110
AD3113	ACCOUNT PROGRAMMING	D21	000270	2	870901	880315	AD3110
MA2100	MFG AUTOMATION	D11	000060	12	860324	(blanks)	(blanks)
MA2110	MFG PROGRAMMING	E21	000100	3	870928	880219	MA2100
MA2112	ROBOT DESIGN	E01	000050	3	860106	881111	MA2110
MA2113	PROD CONTROL PROG	D11	000060	3	(blanks)	(blanks)	MA2100

Employee Project Account Table (TEMPRACT)

EMPNO	PROJNO	ACTNO	STARTDATE	ENDDATE	EMPTIME
000160	MA2100	20	860501	860829	500
000170	MA2100	20	860901	861231	500
000180	MA2100	20	870105	870430	650
000060	MA2100	10	870101	881101	500
000110	MA2100	20	880101	880301	400
000220	MA2112	50	871001	880615	900
000170	MA2112	70	870601	880102	100
000190	MA2112	70	880201	880601	100
000180	MA2113	70	870401	871215	400
000210	MA2113	80	870401	871215	500
000230	MA2113	70	870401	871215	300
000010	AD3100	10	880101	880701	500
000070	AD3110	10	880101	880201	100
000230	AD3111	60	880101	880315	100
000240	AD3111	70	880215	880915	500
000250	AD3112	60	880101	880201	100
000270	AD3113	60	880301	880401	100
000260	AD3113	70	880615	880701	80

Appendix B. SQLCODES

SQL does not communicate directly with the end user but rather returns error codes to the application program when an error occurs.

This appendix lists only SQLCODES. There are many SQL messages, but they are not listed here. Detailed descriptions of all SQL/400 messages, including SQLCODEs, are available online and can be displayed and printed from the Display Message Description display. You can access this display by using the Display Message Description (DSPMSGD) CL command.

SQLCODEs are returned in the SQLCODE variable.

Every SQLCODE has a corresponding message in message file QSQLMSG in library QSYS. The message ID for any SQLCODE is constructed by appending the absolute value (4 digits, padded with zeros) of the SQLCODE to SQL. For example, the message ID corresponding to SQLCODE −204 would be SQL0204. The replacement text for each SQL message is stored in SQLERRM in the SQLCA. The SQLCA is an area in the application program (defined by the application program) for the use of SQL. It is described in *SQL/400 Reference* manual.

If SQL encounters an error while processing the statement, the SQLCODE is a *negative* number. If SQL encounters an exceptional but valid condition while processing your statement, the SQLCODE is a *positive* number. If your SQL statement is processed without encountering an error or exceptional condition, the return code is zero.

When running in debug mode, SQL places a message corresponding to the SQLCODE in the job log for each SQL statement run.

An application may also send the SQL message corresponding to any SQLCODE by specifying the message ID and the replacement text on the Retrieve Message (RTVMSG), Send Program Message (SNDPGMMSG), and Send User Message (SNDUSRMSG) CL commands. For more information on the CL commands, see the *CL Reference*.

© Copyright IBM Corp. 1988, 1989

SQLCODE Descriptions

In the following brief descriptions of the SQLCODEs, message data fields are identified by an ampersand (&) and a number (for example, &1). The replacement text for these fields is stored in SQLERRM in the SQLCA. More detailed cause and recovery information for any SQLCODE can be found by using the Display Message Description (DSPMSGD) CL command.

Positive SQLCODEs

N/A **SQLCODE 0**

Explanation: The SQL statement has run successfully. Check SQLWARN0 to ensure that it is blank. If it is blank, the statement was run successfully. If it is not blank, a warning condition exists. Check the other warning indicators to determine the particular warning condition. For example, if SQLWARN1 is not blank, a string has been truncated.

SQL0100 **SQLCODE +100**

Explanation: Row not found.

SQL0304 **SQLCODE +304**

Explanation: Conversion error in assignment to host variable &1.

SQL0802 **SQLCODE +802**

Explanation: Data conversion or data mapping error.

Negative SQLCODEs

SQL0010 **SQLCODE -10**

Explanation: String constant beginning &1 not delimited.

SQL0060 **SQLCODE -60**

Explanation: Value &1 for DECIMAL function not valid.

SQL0084 **SQLCODE -84**

Explanation: SQL statement not allowed.

SQL0101 **SQLCODE -101**

Explanation: SQL statement too long or complex.

SQL0103 **SQLCODE -103**

Explanation: Numeric literal &1 not valid.

SQL0104 **SQLCODE -104**

Explanation: Token &1 not valid. Valid tokens: &2.

SQL0105 **SQLCODE -105**

Explanation: Mixed string constant not valid.

SQL0106 **SQLCODE -106**

Explanation: Precision specified for FLOAT column not valid.

SQL0107 **SQLCODE -107**

Explanation: &1 too long. Maximum &2 characters.

SQL0109 **SQLCODE -109**

Explanation: &1 clause not allowed.

SQL0110 **SQLCODE -110**

Explanation: Hex literal beginning &1 not valid.

SQL0111 **SQLCODE -111**

Explanation: Column function does not include column name.

SQL0112 **SQLCODE -112**

Explanation: Operand of column function is another function.

SQL0113 **SQLCODE -113**

Explanation: Name &1 not allowed.

SQL0115 **SQLCODE -115**

Explanation: Comparison operator &1 not valid.

SQL0117 **SQLCODE -117**

Explanation: Statement inserts wrong number of values.

SQL0118 **SQLCODE -118**

Explanation: Table &1 in &2 also specified in FROM clause.

SQL0119 **SQLCODE -119**

Explanation: Column &1 in HAVING clause not in GROUP BY.

SQL0120 **SQLCODE -120**

Explanation: Column function use not valid.

SQL0121 **SQLCODE -121**

Explanation: Duplicate column name &1 in INSERT or UPDATE.

SQL0122 **SQLCODE -122**

Explanation: Column specified in SELECT list not valid.

SQL0125 **SQLCODE -125**

Explanation: ORDER BY column number &1 not valid.

SQL0129 **SQLCODE -129**

Explanation: Too many tables in SQL statement.

SQL0131 **SQLCODE -131**

Explanation: Comparison operator LIKE has operands not compatible.

SQL0132 **SQLCODE -132**

Explanation: LIKE predicate not valid.

SQL0134 **SQLCODE -134**

Explanation: Operand of MIN or MAX function exceeds 256 bytes.

SQL0136 **SQLCODE -136**

Explanation: ORDER BY or GROUP BY columns too long.

SQL0137 **SQLCODE -137**

Explanation: Result of concatenation too long.

SQL0138 **SQLCODE -138**

Explanation: Argument &1 of SUBSTR function out of range.

SQL0150 **SQLCODE -150**

Explanation: View or logical file &1 in &2 read only.

SQL0151 **SQLCODE -151**

Explanation: Column &3 in table &1 in &2 read only.

SQL0153 **SQLCODE -153**

Explanation: Column list required for CREATE VIEW.

SQL0154 **SQLCODE -154**

Explanation: UNION and UNION ALL for CREATE VIEW not valid.

SQL0155 **SQLCODE -155**

Explanation: View &1 in &2 not valid in FROM clause.

SQL0156 **SQLCODE -156**

Explanation: &1 in &2 not a table.

SQL0158 **SQLCODE -158**

Explanation: Number of columns specified not consistent.

SQL0159 **SQLCODE -159**

Explanation: &1 in &2 not correct type.

SQL0170 **SQLCODE -170**

Explanation: Number of arguments for function &1 not valid.

SQL0171 **SQLCODE -171**

Explanation: Argument of function &2 is not valid.

SQL0198 **SQLCODE -198**

Explanation: SQL statement empty or blank.

SQL0199 **SQLCODE -199**

Explanation: Keyword &1 not expected. Valid tokens: &2.

SQL0203 **SQLCODE -203**

Explanation: Column &1 in more than one table.

SQL0204 **SQLCODE -204**

Explanation: Object &1 in &2 type *&3 not found.

SQL0205 **SQLCODE -205**

Explanation: Column &1 not in table &2.

SQL0206 **SQLCODE -206**

Explanation: Column &1 not in specified tables.

SQL0207 **SQLCODE -207**

Explanation: ORDER BY clause with UNION operator not valid.

SQL0208 **SQLCODE -208**

Explanation: ORDER BY column &1 not in results table.

SQL0301 **SQLCODE -301**

Explanation: Input host variable &1 not valid.

SQL0302 **SQLCODE -302**

Explanation: Conversion error on input host variable &1.

SQL0303 **SQLCODE -303**

Explanation: Host variable &1 not compatible with SELECT item.

SQL0304 **SQLCODE -304**

Explanation: Conversion error in assignment to host variable &1.

SQL0305 **SQLCODE -305**

Explanation: Indicator variable required.

SQL0309 **SQLCODE -309**

Explanation: Indicator variable contains null value.

SQL0311 **SQLCODE -311**

Explanation: Variable length host variable contains invalid length.

SQL0312 **SQLCODE -312**

Explanation: Host variable &1 not defined or not usable.

SQL0313 **SQLCODE -313**

Explanation: Number host variables not valid.

SQL0401 **SQLCODE -401**

Explanation: Comparison operator &1 operands not compatible.

SQL0402 **SQLCODE -402**

Explanation: &1 use not valid.

SQL0404 **SQLCODE -404**

Explanation: Value for column &1 too long.

SQL0405 **SQLCODE -405**

Explanation: Literal &1 out of range.

SQL0406 **SQLCODE -406**

Explanation: Conversion error on assignment to column &1.

SQL0407 **SQLCODE -407**

Explanation: INSERT or UPDATE value is null.

SQL0408 **SQLCODE -408**

Explanation: INSERT or UPDATE value for column &1 not compatible.

SQL0410 **SQLCODE -410**

Explanation: Floating point literal &1 not valid.

SQL0414 **SQLCODE -414**

Explanation: Numeric column &1 not valid in LIKE predicate.

SQL0415 **SQLCODE -415**

Explanation: UNION operands not compatible.

SQL0417 **SQLCODE -417**

Explanation: Combination of parameter markers not valid.

SQL0418 **SQLCODE -418**

Explanation: Use of parameter marker is not valid.

SQL0419 **SQLCODE -419**

Explanation: Negative scale not valid.

SQL0421 **SQLCODE -421**

Explanation: Number of UNION operands not equal.

SQL0501 **SQLCODE -501**

Explanation: Cursor &1 not open.

SQL0502 **SQLCODE -502**

Explanation: Cursor &1 already open.

SQL0503 **SQLCODE -503**

Explanation: Column &3 cannot be updated.

SQL0504 **SQLCODE -504**

Explanation: Cursor &1 not declared.

SQL0507 **SQLCODE -507**

Explanation: Cursor &1 not open.

SQL0508 **SQLCODE -508**

Explanation: Cursor &1 not positioned on locked row.

SQL0509 **SQLCODE -509**

Explanation: Table &2 in &3 not same as table in cursor &1.

SQL0510 **SQLCODE -510**

Explanation: View or logical file &1 in &2 read only.

SQL0511 **SQLCODE -511**

Explanation: FOR UPDATE OF clause not valid.

SQL0514 **SQLCODE -514**

Explanation: Prepared statement &2 not found.

SQL0516 **SQLCODE -516**

Explanation: Prepared statement &1 not found.

SQL0517 **SQLCODE -517**

Explanation: Prepared statement &2 not SELECT statement.

SQL0518 **SQLCODE -518**

Explanation: Prepared statement &1 not found.

SQL0519 **SQLCODE -519**

Explanation: Prepared statement &1 in use.

SQL0551 **SQLCODE -551**

Explanation: Not authorized to object &1 in &2 type *&3.

SQL0552 **SQLCODE -552**

Explanation: Not authorized to &1.

SQL0556 **SQLCODE -556**

Explanation: Revoke of privilege not valid.

SQL0601 **SQLCODE -601**

Explanation: Object &1 in &2 type *&3 already exists.

SQL0602 **SQLCODE -602**

Explanation: More than 120 columns specified for CREATE INDEX.

SQL0603 **SQLCODE -603**

Explanation: Unique index cannot be created because of duplicate keys.

SQL0604 **SQLCODE -604**

Explanation: Attributes of column not valid.

SQL0607 **SQLCODE -607**

Explanation: Operation not allowed on system table &1 in &2.

SQL0612 **SQLCODE -612**

Explanation: &1 is a duplicate column name.

SQL0614 **SQLCODE -614**

Explanation: Length of columns for CREATE INDEX too long.

SQL0637 **SQLCODE -637**

Explanation: Duplicate &1 keyword.

SQL0802 **SQLCODE -802**

Explanation: Data conversion or data mapping error.

SQL0803 **SQLCODE -803**

Explanation: Duplicate key value specified.

SQL0804 **SQLCODE -804**

Explanation: SQLDA not valid.

SQL0811 **SQLCODE -811**

Explanation: Embedded SELECT resulted in more than one row.

SQL0822 **SQLCODE -822**

Explanation: SQLDA contains address not valid.

SQL0840 **SQLCODE -840**

Explanation: Number of selected items exceeds 8000.

SQL0901 **SQLCODE -901**

Explanation: Function check.

SQL0904 **SQLCODE -904**

Explanation: Resource limit exceeded.

SQL0906 **SQLCODE -906**

Explanation: Operation not performed because previous error.

SQL0913 **SQLCODE -913**

Explanation: Row or object &1 in &2 type *&3 in use.

SQL5001 **SQLCODE -5001**

Explanation: Column qualifier &1 is undefined.

SQL5002 **SQLCODE -5002**

Explanation: Library must be specified for table &1.

SQL5003 **SQLCODE -5003**

Explanation: DDL statement used with commitment control.

SQL5004 **SQLCODE -5004**

Explanation: CREATE VIEW not valid with USER or LENGTH specified.

SQL5005 **SQLCODE -5005**

Explanation: Operator &1 not consistent with operands.

SQL5006 **SQLCODE -5006**

Explanation: Duplicate table designator &1 not valid.

SQL5016 **SQLCODE -5016**

Explanation: Table name &1 not valid for naming option.

SQL5017 **SQLCODE -5017**

Explanation: Too many users specified for GRANT or REVOKE.

SQL5019 **SQLCODE -5019**

Explanation: Empty string operand not valid.

SQL5021 **SQLCODE -5021**

Explanation: FOR UPDATE OF column &1 also in ORDER BY.

SQL5022 **SQLCODE -5022**

Explanation: FOR DATA clause not valid for specified type.

SQL7001 **SQLCODE -7001**

Explanation: File &1 in &2 not database file.

SQL7002 **SQLCODE -7002**

Explanation: Override parameter not valid.

SQL7003 **SQLCODE -7003**

Explanation: File &1 in &2 has more than one format.

SQL7004 **SQLCODE -7004**

Explanation: &1 not a collection.

SQL7005 **SQLCODE -7005**

Explanation: FOR MIXED DATA not allowed.

SQL7006 **SQLCODE -7006**

Explanation: Cannot drop collection &1.

SQL7007 **SQLCODE -7007**

Explanation: COMMIT or ROLLBACK not valid.

SQL7008 **SQLCODE -7008**

Explanation: &1 in &2 not valid for operation.

SQL7009 **SQLCODE -7009**

Explanation: USER specified, but user profile name &1 too long.

SQL7010 **SQLCODE -7010**

Explanation: Logical File &1 not valid for CREATE VIEW.

SQL7011 **SQLCODE -7011**

Explanation: &1 in &2 not table, view, or physical file.

SQL7012 **SQLCODE -7012**

Explanation: Privilege not valid for table or view &1 in &2.

Appendix C. Sample Programs Using SQL/400 Statements

This appendix contains a sample application showing how to code SQL statements in each of the languages supported by SQL/400: COBOL, PL/I, C, and RPG.

The sample application adjusts the estimated employee hours for project numbers starting with MA. The application uses the sample tables in Appendix A.

Each sample program produces the same report, which is shown at the end of this appendix. The first part of the report shows the updated estimated employee hours. The second part shows the previous total and the current total of estimated hours by project.

The following notes apply to all three sample programs:

SQL statements can be entered in upper or lowercase.

1 This host language statement retrieves the external definitions for the SQL table TEMPRACT. These definitions can be used as host variables or as a host structure.

Notes:

1. In RPG, field names in an externally described structure that are longer than 6 characters must be renamed.
2. C does not support the retrieval of external definitions.

2 The SQL INCLUDE SQLCA statement is used to include the SQLCA for PL/I, C, and COBOL programs. For RPG programs, the SQL precompiler automatically places the SQLCA data structure into the source at the end of the I specification section.

3 This SQL WHENEVER statement defines the host language label to which control is passed if an SQLERROR (SQLCODE < 0) occurs in an SQL statement. This WHENEVER SQLERROR statement applies to all the following SQL statements until the next WHENEVER SQLERROR statement is encountered.

4 This SQL UPDATE statement updates the *EMPTIME* column, which contains the estimated hours by the rate in the host variable PERCENTAGE (PERCNT for RPG). The updated rows are those that have MA as the first two characters of the *PROJNO* column.

5 This SQL COMMIT statement commits the changes made by the SQL UPDATE statement. Record locks on all changed rows are released.

Note: The program was precompiled using COMMIT(*CHG).

6 This SQL DECLARE CURSOR statement defines cursor C1, which returns all columns of each row from the TEMPRACT table in which MA are the first two characters of the *PROJNO* column. Rows are returned in ascending order by employee number (*EMPNO* column).

7 This SQL OPEN statement opens cursor C1 so that the rows can be fetched.

8 This SQL WHENEVER statement defines the host language label to which control is passed when all rows are fetched (SQLCODE = 100).

© Copyright IBM Corp. 1988, 1989

9 This SQL FETCH statement returns all columns for cursor C1 and places the returned values into the corresponding elements of the host structure. See Note A.

10 After all rows are fetched, control is passed to this label. The SQL CLOSE statement closes cursor C1.

11 This SQL DECLARE CURSOR statement defines cursor C2, which joins the two tables TEMPRACT and TPROJ. The results are grouped by the columns *PROJNO* and *PRNAME.* The COUNT function returns the number of rows in each group. The SUM functions calculate the total of previously estimated hours and the total of current estimated hours. The ORDER BY 1 clause specifies that rows are retrieved based on the contents of the first results column (*TEMPRACT.PROJNO*).

12 This SQL FETCH statement returns the results columns for cursor C2 and places the returned values into the corresponding elements of the host structure described by the program.

13 This SQL WHENEVER statement with the CONTINUE option causes processing to continue to the next statement regardless if an error occurs on the SQL ROLLBACK statement. Errors are not expected on the SQL ROLLBACK statement; however, this prevents the program from going into a loop if an error does occur.

14 This SQL ROLLBACK statement restores the table to its original condition if an error occurred during the update.

SQL Statements in COBOL Programs

```
5728ST1 R02 M00 891006                        IBM SQL/400       CBLEX                                  89-03-28 15:23:44    Page    1
Source type...............COBOL
Program name..............USER1/CBLEX
Source file...............*LIBL/QLBLSRC
Member....................*PGM
Options...................*SRC      *XREF
Target release............*CURRENT
INCLUDE file..............*LIBL/*SRCFILE
Commit....................*CHG
Generation level..........10
Printer file..............*LIBL/SQLPRT
Text......................*SRCMBRTXT
Source member changed on 89-03-17  11:30:28
```

Figure C-1 (Part 1 of 9). Sample COBOL Program Using SQL Statements

```
5728ST1 R02 M00 891006                          IBM SQL/400       CBLEX                          89-03-28 15:23:44   Page   2
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
     1         IDENTIFICATION DIVISION.                                                         100
     2                                                                                          200
     3         PROGRAM-ID.  CBLEX.                                                              300
     4         ENVIRONMENT DIVISION.                                                            400
     5         CONFIGURATION SECTION.                                                           500
     6         SOURCE-COMPUTER. IBM-AS400.                                                      600
     7         OBJECT-COMPUTER. IBM-AS400.                                                      700
     8         INPUT-OUTPUT SECTION.                                                            800
     9                                                                                          900
    10         FILE-CONTROL.                                                                   1000
    11             SELECT PRINTFILE ASSIGN TO PRINTER-QPRINT                                   1100
    12                ORGANIZATION IS SEQUENTIAL.                                              1200
    13                                                                                         1300
    14         DATA DIVISION.                                                                  1400
    15                                                                                         1500
    16         FILE SECTION.                                                                   1600
    17                                                                                         1700
    18         FD  PRINTFILE                                                                   1800
    19             BLOCK CONTAINS 1 RECORDS                                                    1900
    20             LABEL RECORDS ARE OMITTED.                                                  2000
    21         01  PRINT-RECORD PIC X(132).                                                    2100
    22                                                                                         2200
    23         WORKING-STORAGE SECTION.                                                        2300
    24         77  PROJID PIC X(3).                                                            2400
    25         77  PERCENTAGE PIC S999V99 COMP-3.                                              2500
    26                                                                                         2600
    27        ***************************************************************                  2700
    28        *  Structure for report 1.                                    *                  2800
    29        ***************************************************************                  2900
    30                                                                                         3000
    31         01  RPT1.                                                                       3100
    32  1          COPY DDS-TEMPRACT OF USER1-TEMPRACT.                                        3200
    33                                                                                         3300
    34        ***************************************************************                  3400
    35        *  Structure for report 2.                                    *                  3500
    36        ***************************************************************                  3600
    37                                                                                         3700
    38         01  RPT2.                                                                       3800
    39             15  PROJNO PIC X(6).                                                        3900
    40             15  PROJECT-NAME PIC X(36).                                                 4000
    41             15  EMPLOYEE-COUNT PIC S9(4) COMP-4.                                        4100
    42             15  OLD-TOTAL-TIME PIC S9(6)V99 COMP-3.                                     4200
    43             15  NEW-TOTAL-TIME PIC S9(6)V99 COMP-3.                                     4300
    44                                                                                         4400
    45  2          EXEC SQL                                                                    4500
    46                  INCLUDE SQLCA                                                          4600
    47             END-EXEC.                                                                   4700
    48         77  CODE-EDIT PIC ---99.                                                        4800
    49                                                                                         4900
    50        ***************************************************************                  5000
    51        *  Headers for reports.                                       *                  5100
    52        ***************************************************************                  5200
    53                                                                                         5300
    54         01  RPT1-HEADERS.                                                               5400
    55             05  RPT1-HEADER1 PIC X(132)                                                 5500
    56                       VALUE "UPDATED EMPLOYEE PROJECT ACCOUNT DATA".                    5600
    57             05  RPT1-HEADER2.                                                           5700
    58                 10  FILLER PIC X(10) VALUE "EMPLOYEE".                                  5800
    59                 10  FILLER PIC X(9) VALUE "PROJECT".                                    5900
    60                 10  FILLER PIC X(9) VALUE "ACCOUNT".                                    6000
```

Figure C-1 (Part 2 of 9). Sample COBOL Program Using SQL Statements

```
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
    61                10  FILLER PIC X(104) VALUE "EMPLOYEE".                                6100
    62            05  RPT1-HEADER3.                                                          6200
    63                10  FILLER PIC X(10) VALUE " NUMBER".                                  6300
    64                10  FILLER PIC X(9) VALUE "NUMBER".                                    6400
    65                10  FILLER PIC X(9) VALUE "NUMBER".                                    6500
    66                10  FILLER PIC X(104) VALUE " HOURS".                                  6600
    67        01  RPT2-HEADERS.                                                              6700
    68            05  RPT2-HEADER1.                                                          6800
    69                10  FILLER PIC X(21) VALUE SPACES.                                     6900
    70                10  FILLER PIC X(111)                                                  7000
    71                        VALUE "ACCUMULATED STATISTICS BY PROJECT".                     7100
    72            05  RPT2-HEADER2.                                                          7200
    73                10  FILLER PIC X(9) VALUE "PROJECT".                                   7300
    74                10  FILLER PIC X(38) VALUE SPACES.                                     7400
    75                10  FILLER PIC X(11) VALUE "NUMBER OF".                                7500
    76                10  FILLER PIC X(10) VALUE "PREVIOUS".                                 7600
    77                10  FILLER PIC X(65) VALUE " CURRENT".                                 7700
    78            05  RPT2-HEADER3.                                                          7800
    79                10  FILLER PIC X(9) VALUE "NUMBER".                                    7900
    80                10  FILLER PIC X(38) VALUE "PROJECT NAME".                             8000
    81                10  FILLER PIC X(11) VALUE "EMPLOYEES".                                8100
    82                10  FILLER PIC X(10) VALUE " HOURS".                                   8200
    83                10  FILLER PIC X(65) VALUE "  HOURS".                                  8300
    84        01  RPT1-DATA.                                                                 8400
    85            05  FILLER PIC X VALUE SPACE.                                              8500
    86            05  EMPNO PIC X(6).                                                        8600
    87            05  FILLER PIC XXX VALUE SPACES.                                           8700
    88            05  PROJNO PIC X(6).                                                       8800
    89            05  FILLER PIC X(4) VALUE SPACES.                                          8900
    90            05  ACTNO PIC ZZZ99.                                                       9000
    91            05  FILLER PIC X(3) VALUE SPACES.                                          9100
    92            05  EMPTIME PIC ZZZZ9.99.                                                  9200
    93            05  FILLER PIC X(96) VALUE SPACES.                                         9300
    94        01  RPT2-DATA.                                                                 9400
    95            05  PROJNO PIC X(6).                                                       9500
    96            05  FILLER PIC XXX VALUE SPACES.                                           9600
    97            05  PROJECT-NAME PIC X(36).                                                9700
    98            05  FILLER PIC X(4) VALUE SPACES.                                          9800
    99            05  EMPLOYEE-COUNT PIC ZZZ9.                                               9900
   100            05  FILLER PIC X(5) VALUE SPACES.                                         10000
   101            05  OLD-TOTAL-TIME PIC ZZZZ9.99.                                          10100
   102            05  FILLER PIC XX VALUE SPACES.                                           10200
   103            05  NEW-TOTAL-TIME PIC ZZZZ9.99.                                          10300
   104            05  FILLER PIC X(56) VALUE SPACES.                                        10400
   105                                                                                     10500
   106        PROCEDURE DIVISION.                                                           10600
   107                                                                                     10700
   108        A000-MAIN.                                                                    10800
   109            MOVE 0.06 TO PERCENTAGE.                                                  10900
   110            MOVE "MA%" TO PROJID.                                                     11000
   111            OPEN OUTPUT PRINTFILE.                                                    11100
   112                                                                                     11200
   113       ****************************************************************              11300
   114       * Update the selected projects by the new percentage. If an    *              11400
   115       * error occurs during the updat, ROLLBACK the changes,         *              11500
   116       ****************************************************************              11600
   117                                                                                     11700
   118            EXEC SQL                                                                  11800
   119   3             WHENEVER SQLERROR GO TO E010-UPDATE-ERROR                            11900
   120            END-EXEC.                                                                 12000
```

Figure C-1 (Part 3 of 9). Sample COBOL Program Using SQL Statements

```
5728ST1 R02 M00 891006                         IBM SQL/400       CBLEX                           89-03-28 15:23:44   Page     4
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8    SEQNBR  Last change
   121              EXEC SQL                                                                 12100
   122   4               UPDATE USER1/TEMPRACT                                               12200
   123                     SET EMPTIME = EMPTIME * (1+:PERCENTAGE)                           12300
   124                     WHERE PROJNO LIKE :PROJID                                         12400
   125              END-EXEC.                                                                12500
   126                                                                                       12600
   127         ****************************************************************              12700
   128         *  Commit changes.                                             *              12800
   129         ****************************************************************              12900
   130                                                                                       13000
   131              EXEC SQL                                                                 13100
   132   5               COMMIT                                                              13200
   133              END-EXEC.                                                                13300
   134                                                                                       13400
   135              EXEC SQL                                                                 13500
   136                   WHENEVER SQLERROR GO TO E020-REPORT-ERROR                           13600
   137              END-EXEC.                                                                13700
   138                                                                                       13800
   139         ****************************************************************              13900
   140         *  Report the updated statistics for each employee assigned to*              14000
   141         *  the selected projects.                                      *              14100
   142         ****************************************************************              14200
   143                                                                                       14300
   144         ****************************************************************              14400
   145         *  Write out the header for Report 1.                          *              14500
   146         ****************************************************************              14600
   147                                                                                       14700
   148              write print-record from rpt1-header1                                     14800
   149                    before advancing 2 lines.                                          14900
   150              write print-record from rpt1-header2                                     15000
   151                    before advancing 1 line.                                           15100
   152              write print-record from rpt1-header3                                     15200
   153                    before advancing 2 lines.                                          15300
   154              exec sql                                                                 15400
   155   6               declare c1 cursor for                                               15500
   156                     select *                                                          15600
   157                     from user1/tempract                                               15700
   158                     where tempract.projno like :projid                                15800
   159                     order by empno                                                    15900
   160              end-exec.                                                                16000
   161              EXEC SQL                                                                 16100
   162   7               OPEN C1                                                             16200
   163              END-EXEC.                                                                16300
   164                                                                                       16400
   165              PERFORM B000-GENERATE-REPORT1 THRU B010-GENERATE-REPORT1-EXIT            16500
   166                  UNTIL SQLCODE NOT EQUAL TO ZERO.                                     16600
   167                                                                                       16700
   168          A100-DONE1.                                                                  16800
   169   10         EXEC SQL                                                                 16900
   170                   CLOSE C1                                                            17000
   171              END-EXEC.                                                                17100
   172                                                                                       17200
   173         ****************************************************************              17300
   174         *  For each project selected, generate a report containing the*              17400
   175         *  project number, project name, the old total of employee    *              17500
   176         *  hours, and the new total of employee hours for each        *              17600
   177         *  project.                                                   *              17700
   178         ****************************************************************              17800
   179                                                                                       17900
   180                                                                                       18000
```

Note: 8 and 9 are located on Part 5 of this figure.

Figure C-1 (Part 4 of 9). Sample COBOL Program Using SQL Statements

```
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  SEQNBR  Last change
  181        ****************************************************************        18100
  182        *  Write out the header for Report 2.                          *        18200
  183        ****************************************************************        18300
  184                                                                                 18400
  185                MOVE SPACES TO PRINT-RECORD.                                     18500
  186                WRITE PRINT-RECORD BEFORE ADVANCING 2 LINES.                     18600
  187                WRITE PRINT-RECORD FROM RPT2-HEADER1                             18700
  188                      BEFORE ADVANCING 2 LINES.                                  18800
  189                WRITE PRINT-RECORD FROM RPT2-HEADER2                             18900
  190                      BEFORE ADVANCING 1 LINE.                                   19000
  191                WRITE PRINT-RECORD FROM RPT2-HEADER3                             19100
  192                      BEFORE ADVANCING 2 LINES.                                  19200
  193                                                                                 19300
  194                EXEC SQL                                                         19400
  195     11             DECLARE C2 CURSOR FOR                                        19500
  196                      SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),                  19600
  197                             SUM(EMPTIME/(1.0+:PERCENTAGE)),SUM(EMPTIME)         19700
  198                      FROM USER1/TEMPRACT, USER1/TPROJ                           19800
  199                      WHERE TEMPRACT.PROJNO=TPROJ.PROJNO                         19900
  200                      GROUP BY TEMPRACT.PROJNO, PRNAME                           20000
  201                      HAVING TEMPRACT.PROJNO LIKE :PROJID                        20100
  202                      ORDER BY 1                                                 20200
  203                END-EXEC.                                                        20300
  204                EXEC SQL                                                         20400
  205                    OPEN C2                                                      20500
  206                END-EXEC.                                                        20600
  207                                                                                 20700
  208                PERFORM C000-GENERATE-REPORT2 THRU C010-GENERATE-REPORT2-EXIT    20800
  209                    UNTIL SQLCODE NOT EQUAL TO ZERO.                             20900
  210                                                                                 21000
  211            A200-DONE2.                                                          21100
  212                EXEC SQL                                                         21200
  213                    CLOSE C2                                                     21300
  214                END-EXEC.                                                        21400
  215                                                                                 21500
  216        ****************************************************************        21600
  217        *  All done.                                                   *        21700
  218        ****************************************************************        21800
  219                                                                                 21900
  220            A900-MAIN-EXIT.                                                      22000
  221                CLOSE PRINTFILE.                                                 22100
  222                STOP RUN.                                                        22200
  223                                                                                 22300
  224        ****************************************************************        22400
  225        *  Fetch and write the rows to PRINTFILE.                      *        22500
  226        ****************************************************************        22600
  227                                                                                 22700
  228            B000-GENERATE-REPORT1.                                               22800
  229                EXEC SQL                                                         22900
  230     8              WHENEVER NOT FOUND GO TO A100-DONE1                          23000
  231                END-EXEC.                                                        23100
  232                EXEC SQL                                                         23200
  233     9              FETCH C1 INTO :TEMPRACT                                      23300
  234                END-EXEC.                                                        23400
  235                MOVE CORRESPONDING TEMPRACT TO RPT1-DATA.                        23500
  236                WRITE PRINT-RECORD FROM RPT1-DATA                                23600
  237                      BEFORE ADVANCING 1 LINE.                                   23700
  238                                                                                 23800
  239            B010-GENERATE-REPORT1-EXIT.                                          23900
  240                EXIT.                                                            24000
```

Figure C-1 (Part 5 of 9). Sample COBOL Program Using SQL Statements

```
5728ST1 R02 M00 891006                    IBM SQL/400       CBLEX                                89-03-28 15:23:44   Page    6
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
  241                                                                                        24100
  242          ****************************************************************              24200
  243          *  Fetch and write the rows to PRINTFILE.                      *              24300
  244          ****************************************************************              24400
  245                                                                                        24500
  246           C000-GENERATE-REPORT2.                                                       24600
  247               EXEC SQL                                                                 24700
  248                    WHENEVER NOT FOUND GO TO A200-DONE2                                 24800
  249               END-EXEC.                                                                24900
  250               EXEC SQL                                                                 25000
  251     12             FETCH C2 INTO :RPT2                                                 25100
  252               END-EXEC.                                                                25200
  253               MOVE CORRESPONDING RPT2 TO RPT2-DATA.                                    25300
  254               WRITE PRINT-RECORD FROM RPT2-DATA                                        25400
  255                     BEFORE ADVANCING 1 LINE.                                           25500
  256                                                                                        25600
  257           C010-GENERATE-REPORT2-EXIT.                                                  25700
  258               EXIT.                                                                    25800
  259                                                                                        25900
  260          ****************************************************************              26000
  261          *  Error occured while updating table.  Inform user and        *              26100
  262          *  rollback changes.                                           *              26200
  263          ****************************************************************              26300
  264                                                                                        26400
  265           E010-UPDATE-ERROR.                                                           26500
  266               EXEC SQL                                                                 26600
  267     13             WHENEVER SQLERROR CONTINUE                                          26700
  268               END-EXEC.                                                                26800
  269               MOVE SQLCODE TO CODE-EDIT.                                               26900
  270               STRING "*** ERROR Occurred while updating table.  SQLCODE="              27000
  271                     CODE-EDIT DELIMITED BY SIZE INTO PRINT-RECORD.                     27100
  272               WRITE PRINT-RECORD.                                                      27200
  273               EXEC SQL                                                                 27300
  274     14             ROLLBACK                                                            27400
  275               END-EXEC.                                                                27500
  276               STOP RUN.                                                                27600
  277                                                                                        27700
  278          ****************************************************************              27800
  279          *  Error occured while generating reports.  Inform user and    *              27900
  280          *  exit.                                                       *              28000
  281          ****************************************************************              28100
  282                                                                                        28200
  283           E020-REPORT-ERROR.                                                           28300
  284               MOVE SQLCODE TO CODE-EDIT.                                               28400
  285               STRING "*** ERROR Occurred while generating reports.  SQLCODE            28500
  286          -          "=" CODE-EDIT DELIMITED BY SIZE INTO PRINT-RECORD.                 28600
  287               WRITE PRINT-RECORD.                                                      28700
  288               STOP RUN.                                                                28800
                                * * * * *  E N D  O F  S O U R C E  * * * * *
```

Figure C-1 (Part 6 of 9). Sample COBOL Program Using SQL Statements

```
                                                     CROSS REFERENCE
Data Names                      Define      Reference
'ACTNO'                          122          SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN 'USER1'.'TEMPRACT'
'C1'                             155          CURSOR
                                              162 170 233
'C2'                             195          CURSOR
                                              205 213 251
'DEPTMGR'                        198          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TPROJ'
'DEPTNO'                         198          CHARACTER(3) COLUMN (NOT NULL) IN 'USER1'.'TPROJ'
'EMPNO'                          122          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TEMPRACT'
'EMPNO'                          ****         COLUMN
                                              159
'EMPTIME'                        ****         COLUMN
                                              123 123 197 197
'EMPTIME'                        122          DECIMAL(5,2) COLUMN (NOT NULL) IN 'USER1'.'TEMPRACT'
'ENDDATE'                        122          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TEMPRACT'
'MAJPROJ'                        198          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TPROJ'
'PRENDATE'                       198          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TPROJ'
'PRNAME'                         ****         COLUMN
                                              196 200
'PRNAME'                         198          CHARACTER(36) COLUMN (NOT NULL) IN 'USER1'.'TPROJ'
'PROJNO'                         ****         COLUMN
                                              124
'PROJNO'                         122          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TEMPRACT'
'PROJNO'                         ****         COLUMN IN 'USER1'.'TEMPRACT'
                                              158 196 199 200 201
'PROJNO'                         ****         COLUMN IN 'USER1'.'TPROJ'
                                              199
'PROJNO'                         198          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TPROJ'
'PRSTAFF'                        198          DECIMAL(5,2) COLUMN (NOT NULL) IN 'USER1'.'TPROJ'
'PRSTDATE'                       198          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TPROJ'
'STARTDATE'                      122          CHARACTER(6) COLUMN (NOT NULL) IN 'USER1'.'TEMPRACT'
'TEMPRACT'                       ****         TABLE IN 'USER1'
                                              122 157 158 198
'TPROJ'                          ****         TABLE IN 'USER1'
                                              198 199
'USER1'                          ****         COLLECTION
                                              122 157 198 198
ACTNO                            38           SMALL INTEGER PRECISION(4,0) IN TEMPRACT
ACTNO                            90           IN RPT1-DATA
A100-DONE1                       ****         LABEL
                                              230
A200-DONE2                       ****         LABEL
                                              248
```

Figure C-1 (Part 7 of 9). Sample COBOL Program Using SQL Statements

```
5728ST1 R02 M00 891006                          IBM SQL/400       CBLEX                             89-03-28 15:23:44   Page     8
                                                CROSS REFERENCE
CODE-EDIT                       48
EMPLOYEE-COUNT                  41         SMALL INTEGER PRECISION(4,0) IN RPT2
EMPLOYEE-COUNT                  99         IN RPT2-DATA
EMPNO                           38         CHARACTER(6) IN TEMPRACT
EMPNO                           86         CHARACTER(6) IN RPT1-DATA
EMPTIME                         38         DECIMAL(5,2) IN TEMPRACT
EMPTIME                         92         IN RPT1-DATA
ENDDATE                         38         CHARACTER(6) IN TEMPRACT
E010-UPDATE-ERROR               ****       LABEL
                                           119
E020-REPORT-ERROR               ****       LABEL
                                           136
NEW-TOTAL-TIME                  43         DECIMAL(8,2) IN RPT2
NEW-TOTAL-TIME                  103        IN RPT2-DATA
OLD-TOTAL-TIME                  42         DECIMAL(8,2) IN RPT2
OLD-TOTAL-TIME                  101        IN RPT2-DATA
PERCENTAGE                      25         DECIMAL(5,2)
                                           123 197
PRINT-RECORD                    21         CHARACTER(132)
PROJECT-NAME                    40         CHARACTER(36) IN RPT2
PROJECT-NAME                    97         CHARACTER(36) IN RPT2-DATA
PROJID                          24         CHARACTER(3)
                                           124 158 201
PROJNO                          38         CHARACTER(6) IN TEMPRACT
PROJNO                          39         CHARACTER(6) IN RPT2
PROJNO                          88         CHARACTER(6) IN RPT1-DATA
PROJNO                          95         CHARACTER(6) IN RPT2-DATA
RPT1                            31
RPT1-DATA                       84
RPT1-HEADERS                    54
RPT1-HEADER1                    55         CHARACTER(132) IN RPT1-HEADERS
RPT1-HEADER2                    57         IN RPT1-HEADERS
RPT1-HEADER3                    62         IN RPT1-HEADERS
RPT2                            38         STRUCTURE
                                           251
RPT2-DATA                       94
RPT2-HEADERS                    67
RPT2-HEADER1                    68         IN RPT2-HEADERS
```

Figure C-1 (Part 8 of 9). Sample COBOL Program Using SQL Statements

```
5728ST1 R02 M00 891006                          IBM SQL/400       CBLEX                             89-03-28 15:23:44   Page     9
                                                CROSS REFERENCE
RPT2-HEADER2                    72         IN RPT2-HEADERS
RPT2-HEADER3                    78         IN RPT2-HEADERS
STARTDATE                       38         CHARACTER(6) IN TEMPRACT
TEMPRACT                        38         STRUCTURE IN RPT1
No errors found in source
  288 Source records processed
                               * * * * *  E N D   O F   L I S T I N G  * * * * *
```

Figure C-1 (Part 9 of 9). Sample COBOL Program Using SQL Statements

SQL Statements in PL/I Programs

```
5728ST1 R02 M00 891006                          IBM SQL/400       PLIEX                                89-03-28 15:25:24   Page   1
Source type...............PLI
Program name..............USER1/PLIEX
Source file...............*LIBL/QPLISRC
Member....................*PGM
Options...................*SRC      *XREF
INCLUDE file..............*LIBL/*SRCFILE
Commit....................*CHG
Generation level..........10
Margins...................*SRCFILE
Printer file..............*LIBL/SQLPRT
Text......................*SRCMBRTXT
Source member changed on 89-03-17  11:29:01
```

Figure C-2 (Part 1 of 6). Sample PL/I Program Using SQL Statements

```
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  SEQNBR  Last change
   1     PLIEX: PROC;                                                                         100
   2                                                                                          200
   3       DCL PROJID CHAR(3);                                                                300
   4       DCL PERCENTAGE FIXED DECIMAL(5,2);                                                 400
   5                                                                                          500
   6       /* File declaration for sysprint */                                                600
   7       DCL SYSPRINT FILE EXTERNAL OUTPUT STREAM PRINT;                                    700
   8                                                                                          800
   9       /* Structure for report 1 */                                                       900
  10       DCL 1 RPT1,                                                                       1000
  11   1         %INCLUDE TEMPRACT(TEMPRACT,RECORD);                                         1100
  12                                                                                         1200
  13       /* Structure for report 2 */                                                      1300
  14       DCL 1 RPT2,                                                                       1400
  15             15 PROJNO CHAR(6),                                                          1500
  16             15 PROJECT_NAME CHAR(36),                                                   1600
  17             15 EMPLOYEE_COUNT FIXED BIN(15),                                            1700
  18             15 OLD_TOTAL_TIME FIXED DECIMAL(8,2),                                       1800
  19             15 NEW_TOTAL_TIME FIXED DECIMAL(8,2);                                       1900
  20                                                                                         2000
  21   2   EXEC SQL INCLUDE SQLCA;                                                           2100
  22                                                                                         2200
  23       PERCENTAGE = 0.06;                                                                2300
  24       PROJID = 'MA%';                                                                   2400
  25       OPEN FILE(SYSPRINT);                                                              2500
  26                                                                                         2600
  27       /* Update the selected projects by the new percentage. If an error */            2700
  28       /* occurs during the update, ROLLBACK the changes.                 */            2800
  29   3   EXEC SQL WHENEVER SQLERROR GO TO UPDATE_ERROR;                                    2900
  30       EXEC SQL                                                                          3000
  31   4        UPDATE USER1/TEMPRACT                                                        3100
  32              SET EMPTIME = EMPTIME * (1+:PERCENTAGE)                                    3200
  33              WHERE PROJNO LIKE :PROJID;                                                 3300
  34                                                                                         3400
  35       /* Commit changes */                                                              3500
  36   5   EXEC SQL                                                                          3600
  37            COMMIT;                                                                      3700
  38       EXEC SQL WHENEVER SQLERROR GO TO REPORT_ERROR;                                    3800
  39                                                                                         3900
  40       /* Report the updated statistics for each employee assigned to the */            4000
  41       /* selected projects.                                               */            4100
  42                                                                                         4200
  43       /* Write out the header for Report 1 */                                           4300
  44       put file(sysprint) edit('UPDATED EMPLOYEE PROJECT ACCOUNT DATA')                  4400
  45                          (col(1),a);                                                    4500
  46       put file(sysprint)                                                                4600
  47           edit('EMPLOYEE','PROJECT','ACCOUNT','EMPLOYEE')                               4700
  48               (skip(2),col(1),a,col(11),a,col(20),a,col(29),a);                         4800
  49       put file(sysprint)                                                                4900
  50           edit('NUMBER','NUMBER','NUMBER','HOURS')                                      5000
  51               (skip,col(2),a,col(11),a,col(20),a,col(30),a,skip);                       5100
  52                                                                                         5200
  53       exec sql                                                                          5300
  54   6        declare c1 cursor for                                                        5400
  55              select *                                                                   5500
  56              from user1/tempract                                                        5600
  57              where tempract.projno like :projid                                         5700
  58              order by empno;                                                            5800
  59       EXEC SQL                                                                          5900
  60   7        OPEN C1;                                                                     6000
```

Figure C-2 (Part 2 of 6). Sample PL/I Program Using SQL Statements

```
5728ST1 R02 M00 891006                          IBM SQL/400       PLIEX                             89-03-28 15:25:24   Page      3
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
    61                                                                                         6100
    62        /* Fetch and write the rows to SYSPRINT */                                       6200
    63 8      EXEC SQL WHENEVER NOT FOUND GO TO DONE1;                                         6300
    64                                                                                         6400
    65        DO UNTIL (SQLCODE ¬= 0);                                                         6500
    66          EXEC SQL                                                                       6600
    67 9             FETCH C1 INTO :RPT1;                                                      6700
    68          PUT FILE(SYSPRINT)                                                             6800
    69              EDIT(RPT1.EMPNO,RPT1.PROJNO,RPT1.ACTNO,RPT1.EMPTIME)                       6900
    70                  (SKIP,COL(2),A,COL(11),A,COL(21),F(5),COL(29),F(8,2));                 7000
    71        END;                                                                             7100
    72                                                                                         7200
    73      DONE1:                                                                             7300
    74 10     EXEC SQL                                                                         7400
    75             CLOSE C1;                                                                   7500
    76                                                                                         7600
    77        /* For each project selected, generate a report containing the     */            7700
    78        /* project number, project name, the old total of employee hours,  */            7800
    79        /* and the new total of employee hours for each project.           */            7900
    80                                                                                         8000
    81        /* Write out the header for Report 2 */                                          8100
    82        PUT FILE(SYSPRINT) EDIT('ACCUMULATED STATISTICS BY PROJECT')                     8200
    83                           (SKIP(3),COL(22),A);                                          8300
    84        PUT FILE(SYSPRINT)                                                               8400
    85            EDIT('PROJECT','NUMBER OF','PREVIOUS','CURRENT')                             8500
    86                (SKIP(2),COL(1),A,COL(48),A,COL(59),A,COL(70),A);                        8600
    87        PUT FILE(SYSPRINT)                                                               8700
    88            EDIT('NUMBER','PROJECT NAME','EMPLOYEES','HOURS','HOURS')                    8800
    89                (SKIP,COL(1),A,COL(10),A,COL(48),A,COL(60),A,COL(71),                    8900
    90                 A,SKIP);                                                                9000
    91                                                                                         9100
    92        EXEC SQL                                                                         9200
    93 11          DECLARE C2 CURSOR FOR                                                       9300
    94               SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),                                 9400
    95                  SUM(EMPTIME/(1.0+:PERCENTAGE)),SUM(EMPTIME)                            9500
    96               FROM USER1/TEMPRACT, USER1/TPROJ                                          9600
    97               WHERE TEMPRACT.PROJNO=TPROJ.PROJNO                                        9700
    98               GROUP BY TEMPRACT.PROJNO, PRNAME                                          9800
    99               HAVING TEMPRACT.PROJNO LIKE :PROJID                                       9900
   100               ORDER BY 1;                                                              10000
   101        EXEC SQL                                                                        10100
   102             OPEN C2;                                                                   10200
   103                                                                                        10300
   104        /* Fetch and write the rows to SYSPRINT */                                      10400
   105        EXEC SQL WHENEVER NOT FOUND GO TO DONE2;                                        10500
   106                                                                                        10600
   107        DO UNTIL (SQLCODE ¬= 0);                                                        10700
   108          EXEC SQL                                                                      10800
   109 12            FETCH C2 INTO :RPT2;                                                     10900
   110          PUT FILE(SYSPRINT)                                                            11000
   111              EDIT(RPT2.PROJNO,RPT2.PROJECT_NAME,EMPLOYEE_COUNT,                        11100
   112                   OLD_TOTAL_TIME,NEW_TOTAL_TIME)                                       11200
   113                  (SKIP,COL(1),A,COL(10),A,COL(50),F(4),COL(59),F(8,2),                 11300
   114                  COL(69),F(8,2));                                                      11400
   115        END;                                                                            11500
   116                                                                                        11600
   117      DONE2:                                                                            11700
   118        EXEC SQL                                                                        11800
   119             CLOSE C2;                                                                  11900
   120        GO TO FINISHED;                                                                 12000
```

Figure C-2 (Part 3 of 6). Sample PL/I Program Using SQL Statements

```
5728ST1 R02 M00 891006                          IBM SQL/400       PLIEX                               89-03-28 15:25:24   Page      4
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
  121                                                                                        12100
  122         /* Error occured while updating table.  Inform user and rollback    */         12200
  123         /* changes.                                                         */         12300
  124       UPDATE_ERROR:                                                                    12400
  125  13     EXEC SQL WHENEVER SQLERROR CONTINUE;                                           12500
  126         PUT FILE(SYSPRINT) EDIT('*** ERROR Occurred while updating table.'||           12600
  127           '  SQLCODE=',SQLCODE)(A,F(5));                                               12700
  128         EXEC SQL                                                                       12800
  129  14          ROLLBACK;                                                                 12900
  130         GO TO FINISHED;                                                                13000
  131                                                                                        13100
  132         /* Error occured while generating reports.  Inform user and exit.  */          13200
  133       REPORT_ERROR:                                                                    13300
  134         PUT FILE(SYSPRINT) EDIT('*** ERROR Occurred while generating '||               13400
  135          'reports.  SQLCODE=',SQLCODE)(A,F(5));                                        13500
  136          GO TO FINISHED;                                                               13600
  137                                                                                        13700
  138         /* All done  */                                                                13800
  139       FINISHED:                                                                        13900
  140         CLOSE FILE(SYSPRINT);                                                          14000
  141         RETURN;                                                                        14100
  142                                                                                        14200
  143       END PLIEX;                                                                       14300
                                * * * * *  E N D  O F  S O U R C E  * * * * *
```

Figure C-2 (Part 4 of 6). Sample PL/I Program Using SQL Statements

```
5728ST1 R02 M00 891006                          IBM SQL/400       PLIEX                               89-03-28 15:25:24   Page      5
                                                CROSS REFERENCE
Data Names                    Define    Reference
"ACTNO"                         56        SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"C1"                            54        CURSOR
                                          60 67 75
"C2"                            93        CURSOR
                                          102 109 119
"DEPTMGR"                       96        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"DEPTNO"                        96        CHARACTER(3) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"EMPNO"                         ****      COLUMN
                                          58
"EMPNO"                         56        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"EMPTIME"                       56        DECIMAL(5,2) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"EMPTIME"                       ****      COLUMN
                                          32 32 95 95
"ENDDATE"                       56        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"MAJPROJ"                       96        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRENDATE"                      96        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRNAME"                        ****      COLUMN
                                          94 98
"PRNAME"                        96        CHARACTER(36) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PROJNO"                        ****      COLUMN IN "USER1"."TEMPRACT"
                                          33 57 94 97 98 99
"PROJNO"                        56        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"PROJNO"                        ****      COLUMN IN "USER1"."TPROJ"
                                          97
"PROJNO"                        96        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRSTAFF"                       96        DECIMAL(5,2) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRSTDATE"                      96        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"STARTDATE"                     56        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"TEMPRACT"                      ****      TABLE IN "USER1"
                                          31 56 57 96
"TPROJ"                         ****      TABLE IN "USER1"
                                          96 97
"USER1"                         ****      COLLECTION
                                          31 56 96 96
ACTNO                           11        SMALL INTEGER PRECISION(4,0) IN RPT1
DONE1                           ****      LABEL
                                          63
DONE2                           ****      LABEL
                                          105
EMPLOYEE_COUNT                  17        SMALL INTEGER PRECISION(4,0) IN RPT2
EMPNO                           11        CHARACTER(6) IN RPT1
```

Figure C-2 (Part 5 of 6). Sample PL/I Program Using SQL Statements

```
5728ST1 R02 M00 891006                          IBM SQL/400       PLIEX                                89-03-28 15:25:24   Page     6
                                                     CROSS REFERENCE
EMPTIME                         11        DECIMAL(5,2) IN RPT1
ENDDATE                         11        CHARACTER(6) IN RPT1
NEW_TOTAL_TIME                  19        DECIMAL(8,2) IN RPT2
OLD_TOTAL_TIME                  18        DECIMAL(8,2) IN RPT2
PERCENTAGE                      4         DECIMAL(5,2)
                                          32 95
PROJECT_NAME                    16        CHARACTER(36) IN RPT2
PROJID                          3         CHARACTER(3)
                                          33 57 99
PROJNO                          11        CHARACTER(6) IN RPT1
PROJNO                          15        CHARACTER(6) IN RPT2
REPORT_ERROR                    ****      LABEL
                                          38
RPT1                            10        STRUCTURE
                                          67
RPT2                            14        STRUCTURE
                                          109
STARTDATE                       11        CHARACTER(6) IN RPT1
SYSPRINT                        7
UPDATE_ERROR                    ****      LABEL
No errors found in source
  143 Source records processed
                               * * * * *  E N D  O F  L I S T I N G  * * * * *
```

Figure C-2 (Part 6 of 6). Sample PL/I Program Using SQL Statements

SQL Statements in RPG Programs

```
5728ST1 R02 M00 891006                         IBM SQL/400      RPGEX                               89-03-28 15:24:49    Page    1
Source type...............RPG
Program name..............USER1/RPGEX
Source file...............*LIBL/QRPGSRC
Member....................*PGM
Options...................*SRC      *XREF
Target release............*CURRENT
INCLUDE file..............*LIBL/*SRCFILE
Commit....................*CHG
Generation level..........10
Printer file..............*LIBL/SQLPRT
Text......................*SRCMBRTXT
Source member changed on 89-03-17  11:30:49
```

Figure C-3 (Part 1 of 7). Sample RPG Program Using SQL Statements

```
5728ST1 R02 M00 891006                        IBM SQL/400       RPGEX                                89-03-28 15:24:49    Page   2
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8    SEQNBR  Last change
     1        H                                                                                 100
     2        F*  File declaration for QPRINT                                                   200
     3        F*                                                                                300
     4        FQPRINT  O    F     132           PRINTER                                         400
     5        I*                                                                                500
     6        I* Structure for report 1.                                                        600
     7        I*                                                                                700
     8  1     IRPT1      E DSTEMPRACT                                                           800
     9        I              STARTDATE                        STARDT                            900
    10        I              ENDDATE                          ENDDT                            1000
    11        I              EMPTIME                          EMPTIM                           1100
    12        I*                                                                               1200
    13        I* Structure for report 2.                                                       1300
    14        I*                                                                               1400
    15        IRPT2        DS                                                                  1500
    16        I                                        1   6 PRJNUM                            1600
    17        I                                        7  42 PNAME                             1700
    18        I                                     B  43  440EMPCNT                           1800
    19        I                                     P  45  492OLDTIM                           1900
    20        I                                     P  50  542NEWTIM                           2000
    21        I*                                                                               2100
    22        I            DS                                                                  2200
    23        I                                        1   3 PROJID                            2300
    24  2     I                                     P  4   62PERCNT                            2400
    25        C*                                                                               2500
    26        C                     Z-ADD.06       PERCNT                                      2600
    27        C                     MOVEL'MA%'     PROJID                                      2700
    28        C*                                                                               2800
    29        C* Update the selected projects by the new percentage. If an                     2900
    30        C* error occurs during the update, ROLLBACK the changes.                         3000
    31        C*                                                                               3100
    32  3     C/EXEC SQL WHENEVER SQLERROR GOTO UPDERR                                         3200
    33        C/END-EXEC                                                                       3300
    34        C*                                                                               3400
    35  4     C/EXEC SQL                                                                       3500
    36        C+ UPDATE USER1/TEMPRACT                                                         3600
    37        C+    SET EMPTIME = EMPTIME * (1+:PERCNT)                                        3700
    38        C+    WHERE PROJNO LIKE :PROJID                                                  3800
    39        C/END-EXEC                                                                       3900
    40        C*                                                                               4000
    41        C* Commit changes.                                                               4100
    42        C*                                                                               4200
    43  5     C/EXEC SQL COMMIT                                                                4300
    44        C/END-EXEC                                                                       4400
    45        C*                                                                               4500
    46        C/EXEC SQL WHENEVER SQLERROR GO TO RPTERR                                        4600
    47        C/END-EXEC                                                                       4700
    48        C*                                                                               4800
    49        C* Report the updated statistics for each employee assigned to                   4900
    50        C* selected projects.                                                            5000
    51        C*                                                                               5100
    52        C* Write out the header for report 1.                                            5200
    53        C*                                                                               5300
    54        C                     EXCPTRECA                                                  5400
    55  6     C/EXEC SQL declare c1 cursor for                                                 5500
    56        C+    select * from user1/tempract                                               5600
    57        C+    where tempract.projno like :projid                                         5700
    58        C+    order by empno                                                             5800
    59        C/END-EXEC                                                                       5900
    60        C*                                                                               6000
```

Figure C-3 (Part 2 of 7). Sample RPG Program Using SQL Statements

```
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
  61        C/EXEC SQL                                                                        6100
  62   7    C+  OPEN C1                                                                       6200
  63        C/END-EXEC                                                                        6300
  64        C*                                                                                6400
  65        C* Fetch and write the rows to QPRINT.                                            6500
  66        C*                                                                                6600
  67        C/EXEC SQL WHENEVER NOT FOUND GO TO DONE1                                          6700
  68   8    C/END-EXEC                                                                        6800
  69        C           SQLCOD    DOUNE0                                                      6900
  70   9    C/EXEC SQL                                                                        7000
  71        C+   FETCH C1 INTO :RPT1                                                          7100
  72        C/END-EXEC                                                                        7200
  73        C                     EXCPTRECB                                                   7300
  74        C                     END                                                         7400
  75        C           DONE1     TAG                                                         7500
  76  10    C/EXEC SQL                                                                        7600
  77        C+  CLOSE C1                                                                      7700
  78        C/END-EXEC                                                                        7800
  79        C*                                                                                7900
  80        C* For each project selected, generate a report containing the                    8000
  81        C* project number, project name, the old total of employee hours,                 8100
  82        C* and the new total of employee hours.                                           8200
  83        C*                                                                                8300
  84        C* Write out the header for report 2.                                             8400
  85        C*                                                                                8500
  86        C                     EXCPTRECC                                                   8600
  87  11    C/EXEC SQL                                                                        8700
  88        C+  DECLARE C2 CURSOR FOR                                                         8800
  89        C+    SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),                                   8900
  90        C+       SUM(EMPTIME/(1.0+:PERCNT)),SUM(EMPTIME)                                  9000
  91        C+    FROM USER1/TEMPRACT, USER1/TPROJ                                            9100
  92        C+    WHERE TEMPRACT.PROJNO = TPROJ.PROJNO                                        9200
  93        C+    GROUP BY TEMPRACT.PROJNO, PRNAME                                            9300
  94        C+    HAVING TEMPRACT.PROJNO LIKE :PROJID                                         9400
  95        C+    ORDER BY 1                                                                  9500
  96        C/END-EXEC                                                                        9600
  97        C*                                                                                9700
  98        C/EXEC SQL OPEN C2                                                                9800
  99        C/END-EXEC                                                                        9900
 100        C*                                                                               10000
 101        C* Fetch and write the rows to QPRINT.                                           10100
 102        C*                                                                               10200
 103        C/EXEC SQL WHENEVER NOT FOUND GO TO DONE2                                         10300
 104        C/END-EXEC                                                                       10400
 105        C           SQLCOD    DOUNE0                                                     10500
 106        C/EXEC SQL                                                                       10600
 107  12    C+   FETCH C2 INTO :RPT2                                                         10700
 108        C/END-EXEC                                                                       10800
 109        C                     EXCPTRECD                                                  10900
 110        C                     END                                                        11000
 111        C           DONE2     TAG                                                        11100
 112        C/EXEC SQL CLOSE C2                                                              11200
 113        C/END-EXEC                                                                       11300
 114        C                     GOTO FINISH                                                11400
 115        C*                                                                               11500
 116        C* Error occured while updating table.  Inform user and rollback                 11600
 117        C* changes.                                                                      11700
 118        C*                                                                               11800
 119        C           UPDERR    TAG                                                        11900
 120        C                     EXCPTRECE                                                  12000
```

Figure C-3 (Part 3 of 7). Sample RPG Program Using SQL Statements

```
5728ST1 R02 M00 891006                     IBM SQL/400       RPGEX                       89-03-28 15:24:49   Page     4
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  SEQNBR  Last change
  121  13  C/EXEC SQL WHENEVER SQLERROR CONTINUE                                           12100
  122      C/END-EXEC                                                                      12200
  123      C*                                                                              12300
  124  14  C/EXEC SQL                                                                      12400
  125      C+   ROLLBACK                                                                   12500
  126      C/END-EXEC                                                                      12600
  127      C                    GOTO FINISH                                                12700
  128      C*                                                                              12800
  129      C* Error occured while generating reports.  Inform user and exit.               12900
  130      C*                                                                              13000
  131      C          RPTERR    TAG                                                        13100
  132      C                    EXCPTRECF                                                  13200
  133      C*                                                                              13300
  134      C* All done.                                                                    13400
  135      C*                                                                              13500
  136      C          FINISH    TAG                                                        13600
  137      C                    SETON                     LR                               13700
  138      OQPRINT  E 0201          RECA                                                   13800
  139      O                                  21 'UPDATED EMPLOYEE PROJ'                   13900
  140      O                                  37 'ECT ACCOUNT DATA'                        14000
  141      O        E 01            RECA                                                   14100
  142      O                                   8 'EMPLOYEE'                                14200
  143      O                                  17 'PROJECT'                                 14300
  144      O                                  26 'ACCOUNT'                                 14400
  145      O                                  36 'EMPLOYEE'                                14500
  146      O        E 02            RECA                                                   14600
  147      O                                   7 'NUMBER'                                  14700
  148      O                                  16 'NUMBER'                                  14800
  149      O                                  25 'NUMBER'                                  14900
  150      O                                  34 'HOURS'                                   15000
  151      O        E 01            RECB                                                   15100
  152      O                        EMPNO      7                                           15200
  153      O                        PROJNO    16                                           15300
  154      O                        ACTNO L   26                                           15400
  155      O                        EMPTIML   37                                           15500
  156      O        E 22            RECC                                                   15600
  157      O                                  42 'ACCUMULATED STATISTIC'                   15700
  158      O                                  54 'S BY PROJECT'                            15800
  159      O        E 01            RECC                                                   15900
  160      O                                   7 'PROJECT'                                 16000
  161      O                                  56 'NUMBER OF'                               16100
  162      O                                  66 'PREVIOUS'                                16200
  163      O                                  76 'CURRENT'                                 16300
  164      O        E 02            RECC                                                   16400
  165      O                                   6 'NUMBER'                                  16500
  166      O                                  21 'PROJECT NAME'                            16600
  167      O                                  56 'EMPLOYEES'                               16700
  168      O                                  64 'HOURS'                                   16800
  169      O                                  75 'HOURS'                                   16900
  170      O        E 01            RECD                                                   17000
  171      O                        PRJNUM     6                                           17100
  172      O                        PNAME     45                                           17200
  173      O                        EMPCNTL   55                                           17300
  174      O                        OLDTIML   67                                           17400
  175      O                        NEWTIML   77                                           17500
  176      O        E 01            RECE                                                   17600
  177      O                                  28 '*** ERROR Occurred while'                17700
  178      O                                  52 ' updating table. SQLCODE'                17800
  179      O                                  53 '='                                       17900
  180      O                        SQLCODL   62                                           18000
```

Figure C-3 (Part 4 of 7). Sample RPG Program Using SQL Statements

```
5728ST1 R02 M00 891006                     IBM SQL/400       RPGEX                       89-03-28 15:24:49   Page     5
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  SEQNBR  Last change
  181      O        E 01            RECF                                                   18100
  182      O                                  28 '*** ERROR Occurred while'                18200
  183      O                                  52 ' generating reports. SQL'                18300
  184      O                                  57 'CODE='                                   18400
  185      O                        SQLCODL   67                                           18500
                              * * * * *  E N D  O F  S O U R C E  * * * * *
```

Figure C-3 (Part 5 of 7). Sample RPG Program Using SQL Statements

```
5728ST1 R02 M00 891006                        IBM SQL/400       RPGEX                         89-03-28 15:24:49   Page     6
                                              CROSS REFERENCE
Data Names                    Define    Reference
"ACTNO"                         35        SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"C1"                            55        CURSOR
                                          61 70 76
"C2"                            87        CURSOR
                                          98 106 112
"DEPTMGR"                       87        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"DEPTNO"                        87        CHARACTER(3) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"EMPNO"                         35        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"EMPNO"                         ****      COLUMN
                                          55
"EMPTIME"                       ****      COLUMN
                                          35 35 87 87
"EMPTIME"                       35        DECIMAL(5,2) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"ENDDATE"                       35        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"MAJPROJ"                       87        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRENDATE"                      87        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRNAME"                        ****      COLUMN
                                          87 87
"PRNAME"                        87        CHARACTER(36) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PROJNO"                        ****      COLUMN
                                          35
"PROJNO"                        35        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"PROJNO"                        ****      COLUMN IN "USER1"."TEMPRACT"
                                          55 87 87 87 87
"PROJNO"                        ****      COLUMN IN "USER1"."TPROJ"
                                          87
"PROJNO"                        87        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRSTAFF"                       87        DECIMAL(5,2) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRSTDATE"                      87        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"STARTDATE"                     35        CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"TEMPRACT"                      ****      TABLE IN "USER1"
                                          35 55 55 87
"TPROJ"                         ****      TABLE IN "USER1"
                                          87 87
"USER1"                         ****      COLLECTION
                                          35 55 87 87
ACTNO                           8         SMALL INTEGER PRECISION(4,0) IN RPT1
DONE1                           75        LABEL
                                          67
DONE2                           111       LABEL
                                          103
EMPCNT                          18        SMALL INTEGER PRECISION(4,0) IN RPT2
```

Figure C-3 (Part 6 of 7). Sample RPG Program Using SQL Statements

```
5728ST1 R02 M00 891006                        IBM SQL/400       RPGEX                         89-03-28 15:24:49   Page     7
                                              CROSS REFERENCE
EMPNO                           8         CHARACTER(6) IN RPT1
EMPTIM                          8         DECIMAL(5,2) IN RPT1
ENDDT                           8         CHARACTER(6) IN RPT1
FINISH                          136       LABEL
NEWTIM                          20        DECIMAL(9,2) IN RPT2
OLDTIM                          19        DECIMAL(9,2) IN RPT2
PERCNT                          24        DECIMAL(5,2)
                                          35 87
PNAME                           17        CHARACTER(36) IN RPT2
PRJNUM                          16        CHARACTER(6) IN RPT2
PROJID                          23        CHARACTER(3)
                                          35 55 87
PROJNO                          8         CHARACTER(6) IN RPT1
RPTERR                          131       LABEL
                                          46
RPT1                            8         STRUCTURE
                                          70
RPT2                            15        STRUCTURE
                                          106
STARDT                          8         CHARACTER(6) IN RPT1
UPDERR                          119       LABEL
No errors found in source
  185 Source records processed
                               * * * * *  E N D  O F  L I S T I N G  * * * * *
```

Figure C-3 (Part 7 of 7). Sample RPG Program Using SQL Statements

SQL Statements in C Programs

```
5728ST1 R02 M00 891006                           IBM SQL/400       CEX                                     89-03-28 15:26:00    Page   1
Source type...............C
Program name..............USER1/CEX
Source file...............*LIBL/QCSRC
Member....................*PGM
Options...................*SRC       *XREF
INCLUDE file..............*LIBL/*SRCFILE
Commit....................*CHG
Generation level..........10
Margins...................*SRCFILE
Printer file..............*LIBL/SQLPRT
Text......................*SRCMBRTXT
Source member changed on 89-03-28  12:31:54
```

Figure C-4 (Part 1 of 6). Sample C Program Using SQL Statements

```
5728ST1 R02 M00 891006                         IBM SQL/400       CEX                        89-03-28 15:26:00   Page   2
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
     1  #include "string.h"                                                                  100    89-03-28
     2  #include "stdlib.h"                                                                  200    89-03-28
     3  #include "stdio.h"                                                                   300    89-03-28
     4                                                                                       400    89-03-28
     5  main()                                                                               500    89-03-28
     6  {                                                                                    600    89-03-28
     7     char projid??(3??);                                                               700    89-03-28
     8     double percentage;                                                                800    89-03-28
     9                                                                                       900    89-03-28
    10     /* File declaration for qprint */                                                1000    89-03-28
    11     FILE *qprint;                                                                    1100    89-03-28
    12                                                                                      1200    89-03-28
    13     /* Structure for report 1 */                                                     1300    89-03-28
    14     struct {                                                                         1400    89-03-28
    15             char empno??(7??);                                                       1500    89-03-28
    16             char projno??(7??);                                                      1600    89-03-28
    17             short actno;                                                             1700    89-03-28
    18             char startdate??(7??);                                                   1800    89-03-28
    19             char enddate??(7??);                                                     1900    89-03-28
    20             float emptime;                                                           2000    89-03-28
    21             } rpt1;                                                                  2100    89-03-28
    22                                                                                      2200    89-03-28
    23     /* Structure for report 2 */                                                     2300    89-03-28
    24     struct {                                                                         2400    89-03-28
    25             char projno??(7??);                                                      2500    89-03-28
    26             char project_name??(37??);                                               2600    89-03-28
    27             short employee_count;                                                    2700    89-03-28
    28             double old_total_time,new_total_time;                                    2800    89-03-28
    29            } rpt2;                                                                   2900    89-03-28
    30                                                                                      3000    89-03-28
    31     EXEC SQL INCLUDE SQLCA;                                                          3100    89-03-28
    32                                                                                      3200    89-03-28
    33     percentage = 0.06;                                                               3300    89-03-28
    34     strcpy(projid,"MA%");                                                            3400    89-03-28
    35     qprint=fopen("QPRINT","w");                                                      3500    89-03-28
    36                                                                                      3600    89-03-28
    37     /* Update the selected projects by the new percentage. If an error */            3700    89-03-28
    38     /* occurs during the update, ROLLBACK the changes.                 */            3800    89-03-28
    39     EXEC SQL WHENEVER SQLERROR GO TO update_error;                                   3900    89-03-28
    40     EXEC SQL                                                                         4000    89-03-28
    41          UPDATE USER1/TEMPRACT                                                       4100    89-03-28
    42             SET EMPTIME = EMPTIME * (1+:percentage)                                  4200    89-03-28
    43             WHERE PROJNO LIKE :projid;                                               4300    89-03-28
    44                                                                                      4400    89-03-28
    45     /* Commit changes */                                                             4500    89-03-28
    46     EXEC SQL                                                                         4600    89-03-28
    47          COMMIT;                                                                     4700    89-03-28
    48     EXEC SQL WHENEVER SQLERROR GO TO report_error;                                   4800    89-03-28
    49                                                                                      4900    89-03-28
    50     /* Report the updated statistics for each employee assigned to the */            5000    89-03-28
    51     /* selected projects.                                              */            5100    89-03-28
    52                                                                                      5200    89-03-28
    53     /* Write out the header for Report 1 */                                          5300    89-03-28
    54     fprintf(qprint,     "UPDATED EMPLOYEE PROJECT ACCOUNT DATA");                    5400    89-03-28
    55     fprintf(qprint,"\n\nEMPLOYEE  PROJECT  ACCOUNT  EMPLOYEE");                      5500    89-03-28
    56     fprintf(qprint,  "\n NUMBER   NUMBER   NUMBER    HOURS\n");                      5600    89-03-28
    57                                                                                      5700    89-03-28
    58     exec sql                                                                         5800    89-03-28
    59          declare c1 cursor for                                                       5900    89-03-28
    60            select *                                                                  6000    89-03-28
```

Figure C-4 (Part 2 of 6). Sample C Program Using SQL Statements

```
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
  61              from user1/tempract                                                        6100    89-03-28
  62              where tempract.projno like :projid                                         6200    89-03-28
  63              order by empno;                                                            6300    89-03-28
  64         EXEC SQL                                                                        6400    89-03-28
  65              OPEN C1;                                                                   6500    89-03-28
  66                                                                                         6600    89-03-28
  67         /* Fetch and write the rows to QPRINT */                                        6700    89-03-28
  68         EXEC SQL WHENEVER NOT FOUND GO TO done1;                                        6800    89-03-28
  69                                                                                         6900    89-03-28
  70         do {                                                                            7000    89-03-28
  71           EXEC SQL                                                                      7100    89-03-28
  72                FETCH C1 INTO :rpt1;                                                     7200    89-03-28
  73           fprintf(qprint,"\n %6s   %6s   %6d   %8.2f",                                  7300    89-03-28
  74                   rpt1.empno,rpt1.projno,rpt1.actno,rpt1.emptime);                      7400    89-03-28
  75           }                                                                             7500    89-03-28
  76         while (SQLCODE==0);                                                             7600    89-03-28
  77                                                                                         7700    89-03-28
  78       done1:                                                                            7800    89-03-28
  79         EXEC SQL                                                                        7900    89-03-28
  80              CLOSE C1;                                                                  8000    89-03-28
  81                                                                                         8100    89-03-28
  82         /* For each project selected, generate a report containing the     */           8200    89-03-28
  83         /* project number, project name, the old total of employee hours,  */           8300    89-03-28
  84         /* and the new total of employee hours for each project.           */           8400    89-03-28
  85                                                                                         8500    89-03-28
  86         /* Write out the header for Report 2 */                                         8600    89-03-28
  87         fprintf(qprint,"\n\n\n                    ACCUMULATED STATISTICS\                 8700    89-03-28
  88       BY PROJECT");                                                                     8800    89-03-28
  89         fprintf(qprint,  "\n\nPROJECT                                   \               8900    89-03-28
  90          NUMBER OF  PREVIOUS   CURRENT");                                               9000    89-03-28
  91         fprintf(qprint,    "\nNUMBER   PROJECT NAME                     \               9100    89-03-28
  92          EMPLOYEES   HOURS      HOURS\n");                                              9200    89-03-28
  93                                                                                         9300    89-03-28
  94         EXEC SQL                                                                        9400    89-03-28
  95              DECLARE C2 CURSOR FOR                                                      9500    89-03-28
  96                SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),                                9600    89-03-28
  97                   SUM(EMPTIME/(1.0+:percentage)),SUM(EMPTIME)                           9700    89-03-28
  98                FROM USER1/TEMPRACT, USER1/TPROJ                                         9800    89-03-28
  99                WHERE TEMPRACT.PROJNO=TPROJ.PROJNO                                       9900    89-03-28
 100                GROUP BY TEMPRACT.PROJNO, PRNAME                                        10000    89-03-28
 101                HAVING TEMPRACT.PROJNO LIKE :projid                                     10100    89-03-28
 102                ORDER BY 1;                                                             10200    89-03-28
 103         EXEC SQL                                                                       10300    89-03-28
 104              OPEN C2;                                                                  10400    89-03-28
 105                                                                                        10500    89-03-28
 106         /* Fetch and write the rows to QPRINT */                                       10600    89-03-28
 107         EXEC SQL WHENEVER NOT FOUND GO TO done2;                                       10700    89-03-28
 108                                                                                        10800    89-03-28
 109         do {                                                                           10900    89-03-28
 110           EXEC SQL                                                                     11000    89-03-28
 111                FETCH C2 INTO :rpt2;                                                    11100    89-03-28
 112           fprintf(qprint,"\n%6s    %36s  %6d      %8.2f  %8.2f",                        11200    89-03-28
 113                   rpt2.projno,rpt2.project_name,rpt2.employee_count,                   11300    89-03-28
 114                   rpt2.old_total_time,rpt2.new_total_time);                            11400    89-03-28
 115         }                                                                              11500    89-03-28
 116         while (SQLCODE==0);                                                            11600    89-03-28
 117                                                                                        11700    89-03-28
 118       done2:                                                                           11800    89-03-28
 119         EXEC SQL                                                                       11900    89-03-28
 120              CLOSE C2;                                                                 12000    89-03-28
```

Figure C-4 (Part 3 of 6). Sample C Program Using SQL Statements

```
5728ST1 R02 M00 891006                    IBM SQL/400       CEX                                 89-03-28 15:26:00    Page      4
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  Last change
  121      goto finished;                                                                   12100    89-03-28
  122                                                                                       12200    89-03-28
  123      /* Error occured while updating table.  Inform user and rollback   */             12300    89-03-28
  124      /* changes.                                                       */             12400    89-03-28
  125    update_error:                                                                      12500    89-03-28
  126      EXEC SQL WHENEVER SQLERROR CONTINUE;                                             12600    89-03-28
  127      fprintf(qprint,"*** ERROR Occurred while updating table.  SQLCODE="              12700    89-03-28
  128              "%5d\n",SQLCODE);                                                        12800    89-03-28
  129      EXEC SQL                                                                         12900    89-03-28
  130           ROLLBACK;                                                                   13000    89-03-28
  131      goto finished;                                                                   13100    89-03-28
  132                                                                                       13200    89-03-28
  133      /* Error occured while generating reports.  Inform user and exit.  */            13300    89-03-28
  134    report_error:                                                                      13400    89-03-28
  135      fprintf(qprint,"*** ERROR Occurred while generating reports.  "                  13500    89-03-28
  136              "SQLCODE=%5d\n",SQLCODE);                                                13600    89-03-28
  137      goto finished;                                                                   13700    89-03-28
  138                                                                                       13800    89-03-28
  139      /* All done  */                                                                  13900    89-03-28
  140    finished:                                                                          14000    89-03-28
  141      fclose(qprint);                                                                  14100    89-03-28
  142      exit(0);                                                                         14200    89-03-28
  143                                                                                       14300    89-03-28
  144   }                                                                                   14400    89-03-28
                                * * * * *  E N D  O F  S O U R C E  * * * * *
```

Figure C-4 (Part 4 of 6). Sample C Program Using SQL Statements

```
5728ST1 R02 M00 891006                    IBM SQL/400       CEX                                 89-03-28 15:26:00    Page      5
                                          CROSS REFERENCE
Data Names                     Define   Reference
"ACTNO"                          61       SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"C1"                             59       CURSOR
                                          65 72 80
"C2"                             95       CURSOR
                                          104 111 120
"DEPTMGR"                        98       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"DEPTNO"                         98       CHARACTER(3) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"EMPNO"                          ****     COLUMN
                                          63
"EMPNO"                          61       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"EMPTIME"                        61       DECIMAL(5,2) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"EMPTIME"                        ****     COLUMN
                                          42 42 97 97
"ENDDATE"                        61       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"MAJPROJ"                        98       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRENDATE"                       98       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRNAME"                         ****     COLUMN
                                          96 100
"PRNAME"                         98       CHARACTER(36) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PROJNO"                         ****     COLUMN IN "USER1"."TEMPRACT"
                                          43 62 96 99 100 101
"PROJNO"                         61       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"PROJNO"                         ****     COLUMN IN "USER1"."TPROJ"
                                          99
"PROJNO"                         98       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRSTAFF"                        98       DECIMAL(5,2) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"PRSTDATE"                       98       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TPROJ"
"STARTDATE"                      61       CHARACTER(6) COLUMN (NOT NULL) IN "USER1"."TEMPRACT"
"TEMPRACT"                       ****     TABLE IN "USER1"
                                          41 61 62 98
"TPROJ"                          ****     TABLE IN "USER1"
                                          98 99
"USER1"                          ****     COLLECTION
                                          41 61 98 98
actno                            17       SMALL INTEGER PRECISION(4,0) IN rpt1
done1                            ****     LABEL
                                          68
done2                            ****     LABEL
                                          107
employee_count                   27       SMALL INTEGER PRECISION(4,0) IN rpt2
empno                            15       VARCHAR(7) IN rpt1
```

Figure C-4 (Part 5 of 6). Sample C Program Using SQL Statements

```
5728ST1 R02 M00 891006                               IBM SQL/400       CEX                                         89-03-28 15:26:00   Page     6
                                                     CROSS REFERENCE
emptime                            20           FLOAT(24) IN rptl
enddate                            19           VARCHAR(7) IN rptl
new_total_time                     28           FLOAT(53) IN rpt2
old_total_time                     28           FLOAT(53) IN rpt2
percentage                         8            FLOAT(53)
                                                42 97
project_name                       26           VARCHAR(37) IN rpt2
projid                             7            VARCHAR(3)
                                                43 62 101
projno                             16           VARCHAR(7) IN rptl
projno                             25           VARCHAR(7) IN rpt2
report_error                       ****         LABEL
                                                48
rptl                               21           STRUCTURE
                                                72
rpt2                               29           STRUCTURE
                                                111
startdate                          18           VARCHAR(7) IN rptl
update_error                       ****         LABEL
No errors found in source
  144 Source records processed
                                 * * * * *  E N D   O F   L I S T I N G  * * * * *
```

Figure C-4 (Part 6 of 6). Sample C Program Using SQL Statements

Report Produced by Sample Programs

The following report is produced by each of the preceding sample programs.

```
UPDATED EMPLOYEE PROJECT ACCOUNT DATA

EMPLOYEE  PROJECT  ACCOUNT  EMPLOYEE
 NUMBER   NUMBER   NUMBER    HOURS

 000060   MA2100      10     530.00
 000110   MA2100      20     424.00
 000160   MA2100      20     530.00
 000170   MA2100      20     530.00
 000170   MA2112      70     106.00
 000180   MA2100      20     689.00
 000180   MA2113      70     424.00
 000190   MA2112      70     106.00
 000210   MA2113      80     530.00
 000220   MA2112      50     954.00
 000230   MA2113      70     318.00


                    ACCUMULATED STATISTICS BY PROJECT

PROJECT                                 NUMBER OF  PREVIOUS   CURRENT
NUMBER   PROJECT NAME                   EMPLOYEES   HOURS      HOURS

MA2100   MFG AUTOMATION                     5      2550.00   2703.00
MA2112   ROBOT DESIGN                       3      1100.00   1166.00
MA2113   PROD CONTROL PROG                  3      1200.00   1272.00
```

RSLS765-0

Figure C-5. Report Produced by Sample Programs

Glossary

access path. The path used to locate data specified in SQL statements. An access path can be either indexed or sequential, or a combination of both.

access plan. The control structure produced during compile time that is used to process SQL statements encountered when the program is run.

ANSI. American National Standards Institute

application. A program or set of programs that perform a task; for example, a payroll application.

attribute. In database design, a characteristic of an entity; for example, the telephone number of an employee is one of that employee's attributes.

authorization ID. A user profile. A name identifying a user to whom privileges can be granted.

automatic bind. When an application program is being run and the access plan is not valid, binding takes place automatically; that is, without a user issuing a CRTSQL*xxx* command, where *xxx* is RPG, PLI, CBL, or C.

binary. An SQL data type indicating that the data is a binary number with a precision of 15 (halfword) or 31 (fullword) bits.

bind. The process by which the output from the SQL precompiler is converted to a usable structure called an access plan. This process is the one during which access paths to the data are selected and some authorization checking is performed. There are two types of bind used by SQL/400: automatic and dynamic (see *automatic bind* and *dynamic bind*).

catalog. Tables, maintained by the database manager, that contain descriptions of objects, such as tables, views, and indexes.

catalog views. A set of views containing information about the objects in a collection, such as tables, views, indexes, and column definitions.

character string. A sequence of bytes or characters associated with a single-byte character set.

clause. A distinct part of a statement in the language structure, such as a SELECT clause or a WHERE clause.

collection. A set of objects created by SQL/400 that contains tables, views, indexes, and other system objects (such as a program) created by the user. An SQL collection consists of a library; a data dictionary that contains description and information for all tables, views, indexes, and files created into the library; an SQL catalog; and a journal and journal receiver that are used to journal changes to all tables created into the collection.

column. The vertical part of a table. A column has a name and a particular data type (for example, character, decimal, or integer).

column function. A process that calculates a value from a set of values and expresses it as a function name followed by an argument enclosed in parentheses.

commit. The process that data changed by one application or user to be used by other applications or users. When a commit operation occurs, the locks are released to allow other applications to use the changed data.

commit point. The point in time when data is considered to be consistent.

comparison operator. A symbol (such as =, >, <) used to specify a relationship between two values.

concurrency. The shared use of resources by multiple interactive users or application programs at the same time.

correlation name. An identifier that designates a table, a view, or an individual row of a table or view within a single SQL statement. The name can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

cursor. A named control structure used by an application program to point to a row of data. The position of the row is within a table or view, and the cursor is used to interactively select rows from the columns.

data type. An attribute of columns, constants, and host variables.

DBCS. See *double-byte character set (DBCS)*.

default value. A predetermined value, attribute, or option that is assumed when no other value is explicitly specified. For example, the value of a column is a nonnull value determined by the data type of the column.

delimited identifier. A sequence of one or more characters of the standard character set enclosed within SQL escape characters used to form a name.

delimiter token. A string constant, a delimited identifier, a symbol (for example, ||, /, *, +, or –), or other

© Copyright IBM Corp. 1988, 1989

special characters (for example, period, comma, parentheses).

double-byte character set (DBCS). A set of characters used by national languages, such as Japanese and Chinese, that have more symbols than can be represented by the 256 single-byte EBCDIC positions. Each character is 2 bytes in length, and therefore requires special hardware to be displayed or printed. Contrast with single-byte character set.

dynamic bind. When SQL statements are entered interactively, binding is done dynamically (that is, as the SQL statements are entered).

dynamic SQL. SQL statements that are prepared and processed within a program while the program is running. The SQL source statements are contained in host-language variables rather than being coded directly into the application program. The SQL statement might change several times while the program is running.

EBCDIC. See *extended binary coded decimal interchange code (EBCDIC)*.

embedded SQL. SQL statements that are embedded within a program and are prepared during the program preparation process before the program is run. After it is prepared, the statement itself does not change, although values of host variables specified within the statement might change.

escape character. The symbol used to enclose a delimited identifier. This symbol is the quotation mark ("), except in COBOL programs where the symbol can be assigned by the user as either a quotation mark or an apostrophe.

expression. An operand, or a collection of operators and operands, that yields a single value.

extended binary coded decimal interchange code (EBCDIC). A coded character set of 256 8-bit characters.

fixed-length string. A character string whose length is specified and cannot be changed. Contrast with varying-length string.

fullword binary. A binary number with a precision of 31 bits. See also *integer*.

full select. That form of the select-statement that includes ORDER BY or UNION operators.

function. A column function or a scalar function.

halfword binary. A binary number with a precision of 15 bits.

host language. Any programming language, such as COBOL, PL/I, C, and RPG, in which you can embed SQL statements.

host structure. In an application program, a structure referred to by embedded SQL statements. In RPG, this is called a *data structure*; in PL/I and C, this is known as a *structure*; in COBOL, this is called a *group item*.

host variable. In an application program, a variable referred to by embedded SQL statements. In RPG, this is called a *field name*; in PL/I and C, this is known as a *variable*; in COBOL, this is called a *data item*.

identifier. See *delimited identifier* and *ordinary identifier*.

index. A set of pointers that are logically arranged by the values of a key. Indexes provide quick access to data and can enforce uniqueness on the rows in a table.

index key. The set of columns in a table used to determine the order of indexed entries.

indicator variable. A variable used to represent the null value in an application program. For example, if the value for the results column is null, SQL puts a negative value in the indicator variable.

integer. An SQL data type indicating that the data is a binary number with a precision of 31 bits.

join. A relational operation that allows retrieval of data from two or more tables based on matching column values.

key. A column or an ordered set of columns identified in the description of an index.

keyword. A name that identifies a parameter used in an SQL statement or SQL precompiler command. See also *parameter*.

lock. The process by which integrity of data is ensured. The prevention of concurrent users from accessing inconsistent data.

long string. A string whose actual length, or a varying-length string whose maximum length, is greater than 254 bytes or 127 double-byte characters.

mixed data string. A character string that can contain both single-byte and double-byte characters.

null. A special value that indicates the absence of information.

object. Anything that can be created or manipulated with SQL statements, such as collections, tables, views, or indexes.

ordinary identifier. A letter followed by zero or more characters, each of which is a letter ($, @, #, a-z, and A-Z), a number, or the underscore character used to form a name. An ordinary identifier must not be identical to a reserved word.

ordinary token. A numeric constant, and ordinary identifier, a host variable, or a keyword.

page. A unit of storage equal to 512 bytes.

parameter. The *keywords* and *values* that further define SQL precompiler commands and SQL statements.

plan. See *access plan*.

precompile. A processing of programs containing SQL statements that takes place before a compile. SQL statements are replaced with statements that will be recognized by the host language compiler. The output from this precompile includes source code that can be submitted to the compiler and used in the bind process.

predicate. An element of a search value that expresses or implies a comparison operation.

prepared SQL statement. A named object that is the form of an SQL statement that was processed by the PREPARE statement.

privilege. A capability given to a user by the processing of a GRANT statement.

rebind. The creation a new access plan for a program that was previously bound. If, for example, you add an index for a table that is used by your application program, SQL/400 may automatically bind the application again to take advantage of that index.

real table. A physical file or a table created by SQL.

recovery. The process of rebuilding databases after a system failure.

relational database. A data structure perceived by its users as a collection of tables.

result column. An expression in a SELECT clause that SQL selects for an application program.

result table. The set of rows that SQL selects for an application program. The program uses a cursor to retrieve the rows one by one into a host structure or a set of host variables.

rollback. The process of restoring data changed by an application to the state at its last commit point.

row. The horizontal part of a table. A row consists of a sequence of values, one for each column of the table.

SBCS. See *single-byte character set (SBCS)*.

scalar function. An operation that produces a single value from another value and expresses it in the form of a function name followed by a list of arguments enclosed in parentheses.

search condition. A criterion for selecting rows from a table. A search condition consists of one or more predicates.

short string. A string whose actual length, or a varying-length string whose maximum length, is less than or equal to 254 bytes.

single-byte character set (SBCS). A character set in which each character is represented by a one-byte code.

small integer. An SQL data type indicating that the data is a binary number with a precision of 15 bits.

special register. A storage area whose primary use is to store information produced in conjunction with the use of specific SQL functions. The SQL/400 special register is (named) USER.

SQL. See *Structured Query Language*.

SQLCA. See *SQL communication area (SQLCA)*.

SQLDA. See *SQL descriptor area (SQLDA)*.

SQL communication area (SQLCA). A collection of variables that are used by SQL to provide an application program with information about the processing of SQL statements within the program.

SQL descriptor area (SQLDA). A collection of variables that are used in the processing of certain SQL statements. The SQLDA is intended for dynamic SQL programs.

static SQL. SQL statements that are embedded within a program, and are prepared during the program preparation process before the program is run. After being prepared, the statement itself does not change (although values of host variables specified by the statement might change).

string. A character string.

string delimiter. A symbol used to enclose an SQL string constant. This symbol is the apostrophe ('), except in COBOL applications, in which case the symbol (either an apostrophe or a quotation mark) may be assigned by the user.

Structured Query Language (SQL). A language that can be used within host programming languages or interactively to access data and to control access to resources.

subselect. That form of a query that does not include ORDER BY or UNION operators.

table. A named data object consisting of a specific number of columns and some number of unordered rows.

token. See *delimited token* and *ordinary token*.

union. An SQL operation that combines the results of two subselects. Union is often used to merge lists of values obtained from several tables.

unique index. An index that assures that no identical key values are stored in a table.

unit of recovery. A sequence of operations within a unit of work between two commit points.

unlock. To release an object or system resource that was previously locked and return it to general availability.

user profile. An object with a unique name that contains the user's password, the list of special authorities assigned to a user, and the objects the user owns. See also *authorization ID*.

value. Smallest unit of data manipulated in SQL.

varying-length string. A character string whose length is not fixed, but variable within limits. Contrast with *fixed-length string*.

view. An alternative representation of data from one or more tables. A view can include all or some of the columns contained in the table or tables on which it is defined.

Index

A

abbreviations
- COBOL 5-7
- PL/I 7-7

access plans 10-18
adding to the end of a table 14-10
address variables used in dynamic SQL 9-1
advanced coding techniques
- complex search conditions 3-26
- insert multiple rows into a table 3-32
  - notes on 3-32
- join 3-29
- selecting data from more than one table 3-29

allocating storage 9-10
allocating storage, SQLDA, example 9-10
allowable declarations for host variables
- C 8-6
- COBOL 5-6
- PL/I 7-5
- RPG 6-5

analyzing error and warning messages
- output from the precompiler 10-3

AND keyword 3-26
- multiple search condition 3-27

application plans
- *See* access plans

application requirements
- C
  - communication area (SQLCA) 8-1
- COBOL
  - communication area (SQLCA) 5-1
- PL/I
  - communication area (SQLCA) 7-1
- RPG
  - communication area 6-1

arithmetic expression error 3-6
arranging rows 3-15
arrays, COBOL 5-7
ASP (auxiliary storage pool) 12-3
assignment rules 4-2
- character string assignment 4-3
- COBOL 5-5
- numeric assignment 4-3
- PL/I 7-5
- RPG 6-5

assumptions, relating to examples of SQL statements iii
asterisk (select all columns) 3-7
atomic operations
- data definition statements 12-3
- data integrity 12-3

audience 11-3
authorization
- testing 13-1
- to create test tables
  - GRANT statement example 13-2

authorization ID 12-1
auxiliary storage pool (ASP) 12-3

B

basic SQL statements and clauses 3-1
BETWEEN clause, multiple search condition 3-27
BETWEEN keyword 3-26
binding, access plans 10-18
blocking considerations 14-7
BOTH, PL/I element type 7-8

C

C programs 8-1
- allowable declarations for host variables 8-6
- application requirements 8-1
- coding requirements 8-3
- comments 8-3
- communication area (SQLCA) 8-1
- continuation for SQL statements 8-3
- descriptor area (SQLDA) 8-2
- dynamic SQL 8-2
- host structures 8-4
- host variables 8-4
- indicator variables 8-9
- nulls 8-3
- pointer data types supported 8-8
- preprocessor sequence 8-3
- reserved words 8-3
- SQL statement delimiters 8-3
- SQL syntax in 8-3
- statement labels 8-3
- using SQL statements 8-1

calculation section (RPG) 6-2
catalog description 1-6
catalog views 1-6
- description of 1-6
- SYSCOLUMNS 1-6
- SYSINDEXES 1-8
- SYSKEYS 1-8
- SYSTABLES 1-9
- SYSVIEWDEP 1-9
- SYSVIEWS 1-10

catalogs 1-5, 2-1
- changing information in a table 2-6
- description 1-5
- get information from 2-4
- updating 2-6
- using the catalog in collection design 2-11

© Copyright IBM Corp. 1988, 1989

catalogs *(continued)*
working with 2-1, 2-4, 2-11
changing data 3-3
changing information in a table 2-6
changing the table definition 14-11
changing, indexes 1-5, 2-10
character data type 1-4
character string assignment 4-3
clauses 3-1
AND 3-27
NOT 3-27
OR 3-27
WHERE 3-27
CLOSE CURSOR statement 3-23, 3-25
COBOL programs 5-1
abbreviations 5-7
allowable declarations for host variables 5-6, 5-7
application requirements 5-1
arrays 5-7
assignment rules 5-5
coding requirements 5-2
comments 5-3
communication area (SQLCA) 5-1
continuation for SQL statements 5-3
data description entry
data types 5-7
descriptor area (SQLDA) 5-3
dynamic SQL 5-3
error and warning messages 10-17
during a compile 10-17
host structures 5-4
host variables 5-4
indicator variables 5-8, 7-7
OCCURS clause 5-7
overflow 5-7
procedure division 5-2
requirements for host variables 5-5
reserved words 5-3
return code handling 5-8
sample program with SQL statements C-3
SQL delimiters 5-2
SQL precompiler command example 10-17
SQL syntax in 5-2
truncation 5-7
using SQL statements 5-1
value item 5-7
varying-length character string 5-7
length item 5-7
COBOL source file parameter (QLBLSRC) 10-13
COBOL, external descriptions 5-8
coding examples C-1
SQL statements used in C C-21
SQL statements used in COBOL C-3
SQL statements used in PL/I C-11
SQL statements used in RPG C-16
coding required with host variable, naming conventions 7-4
coding requirements 7-4
C 8-2, 8-3
comments 8-3
dynamic SQL 8-2
host structures 8-4
host variables 8-4
line continuation 8-3
nulls 8-3
preprocessor sequence 8-3
reserved words 8-3
SQL statement delimiters 8-3
COBOL 5-2, 5-3
comments 5-3
dynamic SQL 5-3
host structures 5-4
host variables 5-4, 5-5
including code 5-3
line continuation 5-3
margins 5-3
reserved words 5-3
sequence numbers 5-3
SQL statement delimiters 5-2
PL/I 7-2
comments 7-3
dynamic SQL 7-2
host structures 7-3
host variables 7-3, 7-4
line continuation 7-3
reserved words 7-3
SQL statement delimiters 7-2
RPG 6-2, 6-3
comments 6-3
dynamic SQL 6-3
host variables 6-4
line continuation 6-3
reserved words 6-3
SQL statement delimiters 6-2
coding requirements, RPG, sequence numbers 6-3
coding SQL statements in C 8-1
application requirements 8-1
return code handling 8-9
sample program C-21
coding SQL statements in COBOL 5-1
application requirements 5-1
host structures 5-4
return code handling 5-8
rules for host variables 5-5
sample program C-3
coding SQL statements in PL/I 7-1
application requirements 7-1
host structures 7-9
return code handling 7-11
rules for host variable 7-4
sample program C-11
coding SQL statements in RPG 6-1
application requirements 6-1
host structures 6-4
return code handling 6-7

coding SQL statements in RPG *(continued)*
rules for host variables 6-5
sample program C-16
coding techniques 3-1
collating rows 3-15
collections 1-3
changing list types 11-7
creating 2-1
description 1-3
techniques for solving problems 14-8
terminology 1-2
working with 2-1
columns 1-3
combining 3-29
creating 2-2
data types 1-4
defining column headings 2-2
description 1-3
FOR UPDATE OF clause 3-24
insert default values into columns 3-33
inserting information 2-2
join 3-29
length and data type rules 3-18
numeric, rules for 3-18
selecting all the columns 3-7
specifying the columns you want 3-7
updating in views 2-9
combining information from more than one table 2-5
combining SELECT statements 3-16
commands 10-7, 10-8
CRTSQLC 10-11
CRTSQLCBL 10-8
CRTSQLPLI 10-9
CRTSQLRPG 10-10
for SQL C precompiler (CRTSQLC) 10-11
for SQL COBOL precompiler 10-8
for SQL PL/I precompiler (CRTSQLPLI) 10-9
for SQL RPG precompiler (CRTSQLRPG) 10-10
keywords 10-12
parameter definitions 10-12
SQL precompiler 10-7, 10-12
CRTSQLCBL 10-7
CRTSQLPLI 10-7
CRTSQLRPG 10-7
values 10-12
COMMA, used with PL/I 7-9
COMMENT ON, using 2-13
COMMENT ON, using, getting comments 2-13
comments
C 8-3
COBOL 5-3
PL/I 7-3
RPG 6-3
COMMIT 10-15
interactive keyword 11-18
COMMIT statement 2-3, 2-6, 3-3, 12-4
HOLD value 12-4
commitment control 12-4
changing 11-6
common collection problems, techniques for solving 14-8
communication area (SQLCA)
C 8-1
COBOL 5-1
PL/I 7-1
RPG 6-1
comparison operators 3-9
compiled application program objects 1-10
data dictionary 1-3
program 1-11
temporary source file member 1-11
user source file member 1-10
compiling
application programs 10-17
COBOL 10-17
error and warning messages 10-17
PL/I 10-17
error and warning messages 10-17
RPG 10-17
completing a unit of recovery 3-25
complex search conditions 3-26
keywords for use in search conditions 3-26
multiple search condition 3-27
WHERE 3-27
concatenate rows 3-30
concatenating two sets of selected rows 3-16
concepts 1-1
concepts, SQL with host languages 4-1
assignment rules 4-2
host structures 4-4
indicator variables 4-4
return code, handling 4-6
using host variables 4-1
concurrency, data integrity 12-2
continuation for SQL statements
C 8-3
COBOL 5-3
PL/I 7-3
RPG 6-3
conversion error 3-6
correlation names 2-5, 3-31
CREATE COLLECTION statement 2-1
create SQL statements 10-8, 10-9, 10-10, 10-11
for C precompiler 10-11
for COBOL precompiler (CRTSQLCBL) 10-8
for PL/I precompiler 10-9
for RPG precompiler 10-10
CREATE TABLE statement 2-1, 2-2
CREATE VIEW statement 2-7
creating 2-2
collections 2-1
indexes 2-10
rows 2-2
tables 2-1, 2-2
TDEPT (department table) 2-2

creating *(continued)*
USER1 collection 2-1
creating a view 2-7
combining data from more than one table 2-8
on a single table 2-7
creating a view combining data from more than one table 2-8
creating a view on a single table 2-7
CRTDTADCT CL command 2-1
CRTDUPOBJ CL command 13-2
CRTLIB CL command 2-1
CRTSQLC 10-11
parameter description 10-12
syntax description 10-11
CRTSQLCBL 10-8
parameter description 10-12
syntax description 10-8
CRTSQLCBL command example 10-17
CRTSQLPLI 10-9
parameter description 10-12
syntax description 10-9
CRTSQLRPG 10-10
parameter description 10-12
syntax description 10-10
CURSOR statement 3-25
cursors 3-20
CLOSE CURSOR 3-23
completion of unit of recovery 3-25
defining a cursor 3-22
end of data 3-23
functions 3-20
OPEN CURSOR 3-23
OPEN CURSOR statement with DESCRIPTOR clause 9-15
selecting a set of rows 3-20
setting up a cursor 3-22
using 3-20, 9-14
using parameter markers 9-15

D

damage tolerance 12-6
data definition statement 1-2
data definition statements 12-3
data description entry, COBOL 5-7
data dictionary 1-3
data integrity 12-1, 12-2
atomic operations 12-3
commitment control 12-4
concurrency 12-2
damage tolerance 12-6
data definition statements 12-3
index recovery 12-6
journaling 12-3
save/restore 12-6
data items 5-4
COBOL 5-4
RPG 6-4
data manipulation SQL statement 1-2
data protection 12-1
data integrity 12-2
security 12-1
See also security
data security 12-1
data types 1-4
character 1-4
COBOL 5-7
decimal 1-4
float 1-4
for C applications 8-6, 8-8
for COBOL applications 5-6
for PL/I applications 7-5
for RPG applications 6-5
integer 1-4
numeric 1-4
PL/I 7-7
real 1-4
SMALLINT (small integer) 1-4
DBCS 10-2, 11-17
entering in interactive SQL 11-17
use as precompiler input 10-2
DDM considerations 10-19
deadlock detection 12-2
debugging your program 13-2
decimal data type 1-4
declarations for host variables
C 8-6
COBOL 5-6, 5-7
PL/I 7-5
RPG 6-5
DECLARE CURSOR statement 3-5
defining 3-22
FOR UPDATE OF clause 3-24
selecting several rows at a time 3-5
setting up a cursor 3-22
declare restrictions, PL/I 7-7
defining table name and column headings 2-2
DELETE statement 2-6, 3-4
with WHERE clause 2-6
delete the current row 3-24
DELETE WHERE CURRENT OF statement 3-24
deleting information in a table 2-6
delimiters
for SQL statements in COBOL 5-2
SQL statements in C 8-3
SQL statements in PL/I 7-2
SQL statements in RPG 6-2
department table (TDEPT), creating 2-2
DESCRIBE statement, using with dynamic SQL 9-10
descriptor area (SQLDA)
C 8-2
COBOL 5-3
description 9-7
dynamic SELECT statement format 9-7
fixed-list select-statement 9-5
format description 9-7, 9-8, 9-9
SQLD 9-8

descriptor area (SQLDA) *(continued)*
format description *(continued)*
SQLDABC 9-7
SQLDAID 9-7
SQLDATA 9-9
SQLDIND 9-9
SQLLEN 9-8
SQLN 9-7
SQLNAME 9-9
SQLRES 9-9
SQLTYPE 9-8
SQLVAR 9-8
PL/I 7-2
RPG 6-3
varying-list select-statement 9-5, 9-6
DESCRIPTOR clause with OPEN CURSOR 9-15
designing, test data structure 13-1
dictionary, data 1-3
displaying SQLCODE descriptions B-1
displays 11-3
flow diagram for interactive SQL 11-10
using interactive displays 11-3
DISTINCT keyword, using 3-25
double-byte character set (DBCS) 10-2
entering in interactive SQL 11-17
use as precompiler input 10-2
duplicate rows, eliminating 3-25
dynamic SQL 9-1
allocating storage 9-10
applications 9-1, 9-3
building and running statements 9-3
in C programs 8-2
in COBOL programs 5-3
in PL/I programs 7-2
in RPG programs 6-3
processing 9-3
processing SELECT statements 9-3
processing select-statements 9-5
replacing parameter markers with host variables 9-16
requirements for using parameter markers 9-16
run-time overhead 9-1
statements 9-2
types of statements 9-3
use of address variables 9-1
using a cursor 9-14
using fixed-list 9-5
using OPEN CURSOR with DESCRIPTOR clause 9-15
using parameter markers 9-15
using the DESCRIBE statement 9-10
using the EXECUTE statement 9-4
using the PREPARE statement 9-4
using varying-list 9-5
valid SQL statements 9-1

E

element descriptions, PL/I 7-8, 7-9
COMMA 7-9
element type 7-8
BOTH 7-8
INDICATORS 7-8
INPUT 7-8
KEY 7-8
OUTPUT 7-8
RECORD 7-8
file name 7-8
format name 7-8
prefix name 7-9
element type
used with PL/I 7-8
BOTH 7-8
INDICATORS 7-8
INPUT 7-8
KEY 7-8
OUTPUT 7-8
RECORD 7-8
eliminating duplicate rows 3-25
end of data and cursors 3-23
END-EXEC 6-2
error and warning messages
during a COBOL compile 10-17
during a PL/I compile 10-17
output from the precompiler 10-3
error codes
handling by C 8-9
handling by COBOL 5-8
handling by PL/I 7-11
handling for RPG 6-7
establishing a test environment 13-1
establishing position at the end of a table 14-10
examples iii
assumptions for SQL statements iii
create SQL COBOL command 10-17
using a cursor 3-20
exception condition handling 4-7, 6-7
by RPG 6-7
EXEC SQL 6-2
EXECUTE IMMEDIATE statement, use in PL/I 7-2
EXECUTE statement, using dynamically 9-4
exit interactive SQL 11-8
expressions 3-8
expressions, definition 3-8
external descriptions
COBOL 5-8
RPG 6-6
external file descriptions, PL/I 7-8

F

failed session, recovery 11-9
FETCH statement 14-8
end of data 3-23

FETCH statement *(continued)*
retrieving a row from a set of rows 3-23
file descriptions, external, PL/I 7-8
file name, used with PL/I 7-8
fixed-list, used with dynamic SQL 9-5
float data type 1-4
floating-point number 1-4, 4-3
FOR UPDATE OF clause 3-24
format name, used with PL/I
fullword binary interger 4-3
functional description 11-4
exiting interactive SQL 11-8
interactive SQL display flow diagram 11-10
list selection function 11-1, 11-8, 11-14
session history 11-6
session services 11-1, 11-6
statement entry 11-1, 11-4
supported SQL statements 11-10
function, SQL precompiler 10-1

G

generation level (GENLVL) 10-15
GENLVL 10-15
getting catalog information
about a column 2-11
about a table 2-11
about indexes 2-12
about views 2-12
getting comments 2-13
getting data from more than one table 3-29
getting information
from a single table 2-4
from more than one table 2-5
getting started using interactive SQL 11-3
GO TO label 4-8
GRANT statement 12-1, 13-2
example 13-2
GROUP BY clause 3-11
group of rows 3-13
grouping the rows you select 3-11
GRTOBJAUT (grant object authority) CL command 12-1
guidelines for using SQL 14-1
selecting data from multiple tables, improving performance 14-5
statements 14-1

H

halfword binary integer 4-3
HAVING clause 3-13
help, online 11-9
HOLD value 12-4
host languages, using SQL
host structures 4-4, 5-4
C 8-4
COBOL 5-4

host structures *(continued)*
definition 4-1
in a PL/I program 4-4
PL/I 7-3, 7-9
RPG 6-4
host variables 4-1
See also data items
address variables used in dynamic SQL 9-1
assignment rules for COBOL 5-5
assignment rules for PL/I 7-5
assignment rules for RPG 6-5
C 8-4
C declarations 8-6
COBOL 5-4
COBOL declarations 5-6
input 9-15
INTO clause 4-1
other clauses 4-2
PL/I 7-3
PL/I declarations 7-5
replacing parameter markers with 9-16
requirements 5-5, 7-4
COBOL 5-5
PL/I 7-4
requirements for RPG 6-5
RPG 6-3, 6-4
RPG declarations 6-5
RPG names 6-2
SELECT clause 4-2
using parameter markers 9-15
WHERE clause 4-1
how to, use SQL with host languages

I

IDDU (interactive data definition utility) 1-3
IN clause, multiple search condition 3-27
IN keyword 3-26
INCFILE 10-15
INCFILE parameter 10-2
include file (INCFILE) 10-15
INCLUDE SQLCA statement
C 8-1
COBOL 5-1
PL/I 7-1
RPG 6-1
INCLUDE statement
INCFILE parameter 10-2
use as precompiler input 10-2
including code in a COBOL program 5-3
index recovery 12-6
index search, description of v
indexes 1-5, 2-1
creating 2-10
working with 2-1, 2-10
index, SQL
conditions with out an index 14-3
using effectively 14-3

indicator variables 4-4
C 8-9
COBOL 5-8
PL/I 7-7
RPG 6-6
INDICATORS, PL/I element type 7-8
input data, testing 13-1
input to precompiler 10-2
INPUT, PL/I element type 7-8
insert default values into columns 3-33
insert multiple rows into a table
insert default values into columns 3-33
INSERT statement 2-2, 3-1, A-3
blocking considerations 14-7
using A-3
VALUES clause 3-1
inserting
multiple rows into a table 3-32
rows 2-2
inserting information
columns 2-2
into tables 2-2
rows 2-2
inserting information into tables A-3
inserting multiple rows into a table
notes on 3-32
INTEGER
integer data type 1-4
integrity using views 12-1
integrity, data 12-1
See also security.
interactive SQL
basic functions 11-1
list selection 11-1
session services 11-1, 11-6
statement entry 11-1
statement entry function 11-1, 11-4
display flow diagram 11-10
entering DBCS data 11-17
functional description 11-4
getting started 11-3
help, online 11-9
overview 11-1
STRSQL command 11-18
supported SQL statements 11-10
tips on using interactive SQL 11-14
used for testing SQL statements 11-17
using 11-3
interactive SQL statements 9-2
INTO clause 2-2, 3-5, A-3
used with host variables 4-1
using with INSERT 2-2, A-3
introduction
concepts 1-1
SQL objects description 1-3

J

join 3-29, 3-31
definition 2-5, 2-8
example 2-8
examples 2-5
notes on the technique 3-31
tables 3-29, 3-30
uses for 3-31
WHERE clause 3-30
joining tables 14-5
journal receiver 12-3
journaling 12-3
data integrity 12-3
index function 12-6
*STRJRNAP CL command 12-6
journals and journal receivers 1-3

K

keeping a copy of the data 14-8
keeping duplicates 3-19
keywords
AND 3-26
BETWEEN 3-26
COMMIT 10-15, 11-18
DISTINCT 3-25
for use in search conditions 3-26
GENLVL 10-15
INCFILE (include file) 10-15
LIBOPT 11-19
LIKE 3-26
LISTTYPE 11-19
MARGINS 10-16
NAMING 11-18
NOT keyword 3-10
OPTION 10-13
PGM (program) 10-12
PROCESS 11-19
PRTFILE 10-16
REFRESH 11-19
source file 10-13
source file name 10-13
SRCFILE 10-13
SRCMBR (source member) 10-13
TEXT 10-16
TGTRLS (include file) 10-14
UNION 3-17
UNION ALL 3-19
KEY, PL/I element type 7-8

L

LABEL ON statement 2-2
labels, C 8-3
languages, using SQL with host 4-1
LCKLVL parameter 12-4
left right 10-16

length and data type rules 3-18
LIBOPT, interactive keyword 11-19
library name 10-12, 10-15, 10-16
library, changing list types 11-7
LIKE keyword 3-26
multiple search condition 3-27
line continuation
C 8-3
COBOL 5-3
PL/I 7-3
RPG 6-3
list selection
collection types 11-7
library types 11-7
qualifying names 11-7
list selection function 11-1, 11-8
using interactively 11-14
list type, changing 11-7
listing output from SQL precompiler 10-2
LISTTYPE, interactive keyword 11-19
lock level 12-4
LOCK statement 12-2

M

MARGIN keyword, PL/I 7-3
margins
in a COBOL program 5-3
PL/I 7-3
MARGINS keyword 10-16
use as precompiler input 10-2
member name 10-13
members
source file
changed 10-3
modified 10-3
temporary 10-3
messages 11-9, B-1
during a COBOL compile 10-17
during a PL/I compile 10-17
error and warning
ID B-1
text storage B-1
multiple predicates 3-10
multiple row insertion, notes 3-32

N

naming conventions 1-2
in a PL/I program 7-4
qualifying 11-2, 11-7
SQL naming 11-2
system naming 11-2
NAMING, interactive keyword 11-18
NFYOBJ parameter 12-4
NOT keyword 3-10
multiple search condition 3-27
NOT NULL 2-1, 3-33
NOT NULL WITH DEFAULT 3-33
notes on using a view 2-9
NUL character 8-3
NULL pointer 8-3
null statement 8-3
nulls, C 8-3
numeric
assignment 4-3
columns, rules for 3-18
conversions 14-2
data type 1-4

O

objects
catalogs 1-5
collections 1-3
columns 1-3
compiled application program objects 1-10, 1-11
program 1-11
temporary source file member 1-11
user source file member 1-10
data dictionary 1-3
definitions 1-3
indexes 1-5
journal 1-3
journal receiver 1-3
rows 1-3
SQL 1-3
tables 1-3
views 1-4
OCCURS clause 5-7
online education, description of vi
online information, types of v
help for control language commands vi
help for displays v
index search v
online education vi
question-and-answer function vi
online, help 11-9
OPEN CURSOR statement 3-23, 9-15
using with DESCRIPTOR clause 9-15
open cursors 3-25
OPEN statement 14-6
blocking considerations 14-7
performance considerations 14-6
use of 14-6
operators, comparison 3-9
OPTION keyword 10-13
options
See also keyword
SQL precompiler commands 10-12
OR keyword, multiple search condition 3-27
ORDER BY clause
used when selecting rows 3-14, 3-15
used with DESC and ASC values 3-15

ordering rows 3-15
output
device selection 11-7
from precompiler 10-2
precompiler 10-2, 10-3
listing 10-2
printing current session 11-7
OUTPUT, PL/I element type 7-8
overflow
COBOL 5-7
PL/I 7-7
override considerations 10-19
overview, interactive SQL, audience 11-3

P

paging
through data, previously retrieved 14-8
through retrieved data 14-8
parameter
description 10-12
keyword
COMMIT 10-15
GENLVL (generation level) 10-15
INCFILE (include file) 10-15
MARGINS 10-16
OPTION 10-13
PRTFILE 10-16
SRCMBR (source member) 10-13
TEXT 10-16
TGTRLS 10-14
PGM (program) keyword 10-12
program name 10-12
source file 10-13
SQL precompiler commands 10-12
SRCFILE 10-13
STRSQL 11-18
values
left right 10-16
library name 10-12, 10-15, 10-16
member name 10-13
printer file name 10-16
QLBLSRC 10-13
QPLISRC 10-13
QRPGSRC 10-13
QSYSPRT 10-16
QxxxSRC 10-13
severity level 10-16
source file name 10-13, 10-15
*ALL 10-15
*APOST (apostrophe) 10-14
*APOSTSQL 10-14
*CHG (change) 10-15
*COMMA 10-14
*CURLIB 10-12, 10-15, 10-16
*CURRENT 10-14
*GEN 10-13
*LIBL 10-15, 10-16
*NOGEN 10-13
parameter *(continued)*
values *(continued)*
*NONE 10-15
*NOSOURCE 10-13
*NOSRC 10-13
*NOXREF 10-13
*PERIOD 10-14
*PGM 10-13
*PRV 10-15
*QUOTE 10-14
*QUOTESQL 10-13
*SOURCE 10-13
*SQL 10-14
*SRC 10-13
*SRCFILE (source file) 10-15, 10-16
*SRCMBRTXT (source member text) 10-16
*SYS 10-14
*SYSVAL 10-14
*XREF 10-13
parameter markers 9-15
replacing with host variables 9-16
requirements for using 9-16
using 9-15
PEMPL in a COBOL program 5-5
performance considerations 14-2, 14-5
OPEN statement 14-6
PGM 10-12
PL/I programs 7-1
abbreviations 7-7
allowable declarations for host variables 7-5
application requirements 7-1
assignment rules 7-5
coding requirements 7-2
comments 7-3
communication area (SQLCA) 7-1
continuation for SQL statements 7-3
data types 7-7
declare restrictions 7-7
descriptor area (SQLDA) 7-2
dynamic SQL 7-2
element descriptions 7-8
error and warning messages 10-17
during a compile 10-17
EXECUTE IMMEDIATE statement 7-2
external file descriptions 7-8
host structures 7-3
host variables 7-3
margins 7-3
overflow 7-7
requirements for host variables 7-4
reserved words 7-3
return code handling 7-11, 8-9
sample program with SQL statements C-11, C-16
SQL statement delimiters 7-2
SQL syntax in 7-2
structures 7-9
truncation 7-7
using SQL statements 7-1

PL/I programs *(continued)*
%INCLUDE directive 7-8
PL/I source file parameter (QPLISRC) 10-13
pointer data types, allowed for C programs 8-8
positioning at end of table 14-10
precompiler
basic processes 10-1
commands 10-12
keywords 10-12
parameter definitions 10-12
values 10-12
CRTSQLC 10-11
CRTSQLCBL 10-8
CRTSQLPLI 10-9
CRTSQLRPG 10-10
for C 10-11
for COBOL 10-8
for PL/I 10-9
for RPG 10-10
INCLUDE statement 10-2
input to 10-2
options 10-12
output 10-3
output from 10-2
parameter definitions
library name 10-12
PGM 10-12
program name 10-12
QLBLSRC 10-13
QPLISRC 10-13
QRPGSRC 10-13
QxxxSRC 10-13
source file name 10-13
SRCFILE 10-13
*CURLIB 10-12
SQL commands 10-7, 10-8, 10-9, 10-10, 10-11
temporary source file members 10-3
use of margins 10-2
using double-byte character set (DBCS) 10-2
precompiling SQL statements
precompiling SQL statements, precompile commands 10-7
predicates 3-8, 3-27
in a search condition 3-9
in the WHERE clause 3-8
multiple 3-10
use of 3-8
used in multiple search condition 3-27
prefix name, used with PL/I 7-9
PREPARE statement, using dynamically 9-4
preparing a program containing SQL for running 10-1
print current session 11-7
print file name 10-16
printer file (PRTFILE) 10-16
problem solving, techniques for collection problems 14-8
procedure division (COBOL) 5-2
PROCESS, interactive keyword 11-19
program 1-11
description 1-11
name description 10-12
references 10-18
sample in C C-21
sample in COBOL C-3
sample in PL/I C-11
sample in RPG C-16
prompt function 11-4
protection of data 12-1
PRTFILE 10-16
public authority 12-1

Q

Q & A
See question-and-answer function, description of
QLBLSRC 10-13
QPLISRC 10-13
QRPGSRC 10-13
QSYSPRT 10-16
qualifying
SQL and system names 11-2
SQL statements 3-8
question-and-answer function, description of vi

R

real data type 1-4
RECORD, PL/I element type 7-8
Recover SQL Session display, use of 11-9
recovery 11-9, 12-3
a failed SQL session 11-9
a saved SQL session 11-9
redundant information, providing 14-5
REFRESH, interactive keyword 11-19
register, USER special 3-10
relational database, description 1-2
removing all entries from the current session 11-7
requirements for host variables, indicator variables 7-7
reserved words
C 8-3
COBOL 5-3
PL/I 7-3
RPG 6-3
result table 3-20, 14-9
retrieving
a row from a set of rows 3-23
data a second time 14-8
data from the middle 14-9
from the beginning 14-8
order of rows in the second result table 14-9
in reverse order 14-10
information from a single table 2-4
return codes B-1
exception condition handling 4-7

return codes *(continued)*
handling by C 8-9
handling by COBOL 5-8
handling by PL/I 7-11
handling for RPG 6-7
rules for handling 4-6
revising data 3-3
Revoke Object Authority (RVKOBJAUT) command 12-1
REVOKE statement 12-1
ROLLBACK statement, HOLD value
rows 1-3
arranging 3-15
combining 3-30
creating 2-2
current update 3-24
delete current 3-24
description 1-3
eliminating duplicate 3-25
inserting 2-2
inserting multiple rows 3-32
notes on 3-32
joining 3-30
ordering 3-15
retrieving a row from a set of rows 3-23
specifying a condition for a group of rows 3-13
rows in the second result table, order of 14-9
RPG programs 6-1
allowable declarations for host variables 6-5
application requirements 6-1
assignment rules 6-5
calculation section 6-2
coding requirements 6-2
comments 6-3
communication area (SQLCA) 6-1
continuation for SQL statements 6-3
descriptor area (SQLDA) 6-3
dynamic SQL 6-3
example of SQL statements in RPG 6-3
external descriptions 6-6
host structures 6-4
host variables 6-4
indicator variables 6-6
requirements for host variables 6-5
reserved words 6-3
return code handling 6-7
SQL syntax in 6-2
structures 6-6
using SQL statements 6-1
RPG source file parameter (QRPGSRC) 10-13
rules
assignment rules 4-2
for character string assignment 4-3
for coding host variables in COBOL 5-5
for coding host variables in RPG 6-5
for numeric assignment 4-3
for PL/I host variables 7-4
for SQL with host language 4-6

rules *(continued)*
for SQL with host languages 4-2
host structures 4-4
return code handling 4-6
SQL with host languages 4-4
using SQL with host languages 4-1
running a program containing SQL statements
diagnostic information 10-1
RVKOBJAUT (Revoke Object Authority) command 12-1

S

sample programs C-1
saved session, recovery 11-9
save/restore 12-6
saving session in source file 11-7
search conditions 3-26
using predicates 3-9
security
data 12-1
SQL objects 12-1
using views 12-1
SELECT INTO statement 3-5
select-statements 9-5
allocating storage 9-10
DISTINCT keyword 3-25
dynamic processing 9-3
dynamic SQL 9-3
fixed-list 9-5
processing dynamically 9-5
selecting several rows at a time 3-5
using a cursor 9-14
varying-list 9-5, 9-6
selecting
all the columns 3-7
data from more than one table 3-29
data from multiple tables 14-5
improving performance 14-5
joining tables 14-5
provide redundant information 14-5
one row 3-5
output device 11-7
rows from a table 3-5
sequence numbers
in COBOL 5-3
in RPG 6-3
session history 11-6
session services 11-6
changing collection (library) lists 11-7
changing list types 11-7
commitment control 11-6
controlling SELECT output device 11-7
printing current session 11-7
removing entries from current session 11-7
saving sessions in source file 11-7
statement processing control 11-6

session services function 11-1
sessions 11-9
changing options 11-1
controlling 11-1
functional description 11-9
printing current 11-7
recovery of a failed session 11-9
recovery of a saved session 11-9
removing all entries from the current session 11-7
saving in source file 11-7
session services function 11-1
SET clause 3-3
severity level 10-16
shift-in, shift-out characters, use with interactive SQL 11-17
small integer (SMALLINT) data type 1-4
SMALLINT
SMALLINT data type 1-4
source commands, SQL precompiler 10-7
source file parameters
QLBLSRC (COBOL) 10-13
QPLISRC (PL/I) 10-13
QRPGSRC (RPG) 10-13
source file (SRCFILE) 10-13, 10-16
members 10-3
changed 10-3
modified 10-3
temporary 10-3
name 10-13, 10-15
source member (SRCMBR) 10-13
special register 3-10
specifying the columns you want 3-7
SQL descriptor area (SQLDA), description 9-7
SQL in RPG example 6-3
SQL index, using effectively 14-2
avoid LIKE predicates beginning with % or _ 14-3
avoid numeric conversions 14-2
avoid string character padding 14-2
conditions with out an index 14-3
SQL naming convention 11-2
SQL precompiler, basic processes 10-1
SQL statements iii
assumptions for examples of iii
data definition 1-2
data definition statements 12-3
data manipulation 1-2
types of 1-2
used in sample C program C-21
used in sample COBOL program C-3
used in sample PL/I program C-11
used in sample RPG program C-16
SQL statement, prompting for 11-4
SQL (Structured Query Language), prompting 11-4
SQLCA
C 8-1
COBOL 5-1
PL/I 7-1
RPG 6-1
SQLCODEs B-1
description B-1
how to display descriptions B-1
list B-1
message file B-1
occurrence of 3-6
SQLD 9-8
SQLDA
C 8-2
COBOL 5-3
description 9-7
example of allocating storage 9-10
format description 9-7, 9-8, 9-9
SQLD 9-8
SQLDABC 9-7
SQLDAID 9-7
SQLDATA 9-9
SQLIND 9-9
SQLLEN 9-8
SQLN 9-7
SQLNAME 9-9
SQLRES 9-9
SQLTYPE 9-8
SQLVAR 9-8
PL/I 7-2
RPG 6-3
SQLDABC 9-7
SQLDAID 9-7
SQLDATA 9-9
SQLIND 9-9
SQLLEN 9-8
SQLN 9-7
SQLNAME 9-9
SQLRES 9-9
SQLTYPE 9-8
SQLVAR 9-8
SRCFILE (source file) 10-13
SRCMBR (source member), description 10-13
Start Debug (STRDBG) command 13-2
Start SQL (STRSQL) command 11-18
statement
WHENEVER 8-9
statement entry function 11-1, 11-4
statement processing, controlling 11-6
statements in RPG programs, SQL in RPG example 6-3
statements, SQL 3-1
used in sample C program C-21
used in sample COBOL program C-3
used in sample PL/I program C-11
used in sample RPG program C-16
statement, WHENEVER 5-8, 7-11
static SQL statements 9-2
STRCMTCTL 12-4
STRDBG (Start Debug) CL command 13-2
string, character assignment 4-3
STRSQL (Start SQL) command
description 11-18

STRSQL (Start SQL) command *(continued)*
parameters 11-18
syntax 11-18
structures
COBOL 5-4
halfword integer variables 4-4
indicator 4-4
PL/I 7-9
RPG 6-4, 6-6
RPG data 6-5
subselect
combining two or more statements 3-16
with UNION keyword 3-16
supported SQL statements 11-10
syntax checking, selecting interactively
syntax for SQL precompiler commands 10-7
CRTSQLC 10-11
CRTSQLCBL 10-8
CRTSQLPLI 10-9
CRTSQLRPG 10-10
SYSCOLUMNS 1-6
SYSINDEXES 1-8
SYSKEYS 1-8
SYSTABLES 1-9
system naming convention 11-2
system print file, QSYSPRT parameter 10-16
SYSVIEW 1-10
SYSVIEWDEP 1-9

T

table definitions 14-11
tables 1-3, 2-1
changing information in a table 2-6
CREATE TABLE statement 2-2
creating 2-2
creating tables 2-1
defining table names 2-2
deleting information in a table 2-6
description 1-3
get information from a single table 2-4
getting information from more than one table 2-5
inserting information into tables 2-2, A-3
join 3-31
updating 2-6
using tables 2-1
working with 2-1, 2-2, 2-4, 2-6
tables, views, and catalogs, using COMMENT ON 2-13
target release (TGTRLS) 10-14
TDEPT (department table), creating 2-2
techniques for solving common collection problems 14-8
techniques for solving common database problems
adding to the end of a table 14-10
changing table definitions 14-11
keeping a copy of the data 14-8
positioning at end of table 14-10
retrieving the data a second time 14-8
techniques for solving common database problems *(continued)*
updating data as it is retrieved from a table 14-10
updating data previously retrieved 14-11
techniques for using SQL 14-1
selecting data from multiple tables, improving performance 14-5
statements 14-1
techniques, coding 3-1
temporary source file members 1-11, 10-3
terminology 1-2
collection 1-2
qualifying 11-2
SQL naming 11-2
SQL objects 1-2
system 1-2
system naming 11-2
test data structures 13-1
designing 13-1
test tables 13-2
test data structures, designing, test views of existing tables 13-1
test tables 13-2
test views of existing tables 13-1
testing 13-1
authorization 13-1, 13-2
GRANT statement example 13-2
debugging your program 13-2
establishing a test environment 13-1
SQL statements 13-1
SQL statements interactively 11-17
test data structure 13-1
test data structures 13-1
test input data 13-1
test tables 13-2
TEXT 10-16
TGTRLS 10-14
tips on using interactive SQL 11-14
truncation
COBOL 5-7
PL/I 7-7

U

UNION ALL, keeping duplicates 3-19
UNION keyword 3-16
length and data type rules 3-18
rules for using 3-17, 3-18
unit of recovery
completion of 3-25
description 3-25
UPDATE statement 3-3, 14-10
updating the current row 3-24
using 2-6
WHERE CURRENT OF clause 3-24
updating data
previously retrieved 14-11
retrieved from a database
restrictions 14-10

updating data *(continued)*
retrieved from a table 14-10
updating the current row of a set of rows 3-24
user source file member 1-10
USER special register 3-10
USER1 collection, creating 2-1
using
a view 2-7
catalogs in collection design 2-11
tables 2-1
using a cursor 9-14
using basic SQL statements and clauses 3-1
using interactive 11-3
using SQL statements 14-1
in C programs C-21
in COBOL programs C-3
in PL/I programs C-11
in RPG programs C-16
using an SQL index effectively 14-2

V

validity checking, selecting interactively 11-6
value 10-12
VALUE clause 5-7
value item (COBOL) 5-7
VALUES clause 3-1
values, inserting 3-33
variables 4-1
address variables used in dynamic SQL 9-1
assignment rules for COBOL 5-5
assignment rules for PL/I 7-5
assignment rules for RPG 6-5
C 8-4
C declarations 8-6
C indicator 8-9
COBOL 5-4
COBOL declarations 5-6
COBOL indicator 5-8
halfword integer 4-4
indicator 4-4
input host 9-15
PL/I 7-3
PL/I declarations 7-5
PL/I indicator 7-7
replacing parameter markers with host 9-16
requirements for COBOL 5-5
requirements for PL/I 7-4
requirements for RPG 6-5
RPG 6-4
RPG declarations 6-5
RPG indicator 6-6
RPG names 6-2
using host 4-1
using parameter markers 9-15
varying-length character string
COBOL 5-7
length item 5-7
varying-list
select-statement 9-6
used with dynamic SQL 9-5
views 1-4, 2-1, 3-8
authority to 12-1
catalog
SYSCOLUMNS 1-6
SYSINDEXES 1-8
SYSKEYS 1-8
SYSTABLES 1-9
SYSVIEWDEP 1-9
SYSVIEWS 1-10
changing 2-9
changing information in a table 2-7
creating a view 2-7
defining 2-10
get information from a single view 2-4
getting information from 2-9
inserting information 2-9
processing 2-10, 3-8
running 2-10
updating 2-9
using a view 2-7
working with 2-1, 2-4

W

warning and error messages
during a COBOL compile 10-17
during a PL/I compile 10-17
WHENEVER NOT FOUND clause 3-23
WHENEVER statement 4-7, 5-8, 6-7, 7-11, 8-9
exception condition 4-7
handling 4-7
handling by RPG 6-7
WHERE clause 3-8, 3-27, 3-30, 14-4
AND 3-27
multiple search condition 3-27
NOT 3-27
OR 3-27
used with a join 3-30
used with host variables 4-1
using 2-6
WHERE CURRENT OF clause 3-24
working with
collections 2-1
indexes 2-10

Special Characters

***ALL 10-15**
***APOST 10-14**
***APOSTSQL 10-14**
***CHG 10-15**
***COMMA 10-14**
***CURLIB 10-12, 10-15, 10-16**
***CURRENT 10-14**

***GEN 10-13**
***LIBL 10-15, 10-16**
***NOGEN 10-13**
***NONE 10-15**
***NOSOURCE 10-13**
***NOSRC 10-13**
***NOXREF 10-13**
***PERIOD 10-14**
***PGM (program)**
  description 10-13
***PRV 10-15**
***QUOTE 10-14**
***QUOTESQL 10-13**
***SAVLIB CL command 12-6**
***SAVOBJ CL command 12-6**
***SOURCE 10-1, 10-13**
***SQL 10-14**
***SRC 10-13**
***SRCFILE 10-16**
***SRCFILE (source file) 10-15**
***SRCMBRTXT 10-16**
***STRJRNAP CL command 12-6**
***SYS 10-14**
***SYSVAL 10-14**
***XREF 10-1, 10-13**
%INCLUDE directive 7-8
  example 7-11
  prefix 7-9
  structure ending 7-10
  structures 7-9
  using prefixes 7-9

Application System/400™ Programming:
Structured Query Language/400
Programmer's Guide

SC21-9609-1

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. 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.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): Comment(s):

Please contact your IBM representative or your IBM-approved remarketer to request additional publications.

Name

Company or
Organization

Address

City State Zip Code

Phone No.

Area Code

No postage necessary if mailed in the U.S.A.

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 MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
3605 North Hwy 52
ROCHESTER MN 55901-9986

Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold
Along Line

Application System/400™ Programming:
Structured Query Language/400
Programmer's Guide

SC21-9609-1

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. 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.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): Comment(s):

Please contact your IBM representative or your IBM-approved remarketer to request additional publications.

Name

Company or
Organization

Address

City State Zip Code

Phone No.

Area Code

No postage necessary if mailed in the U.S.A.

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 MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

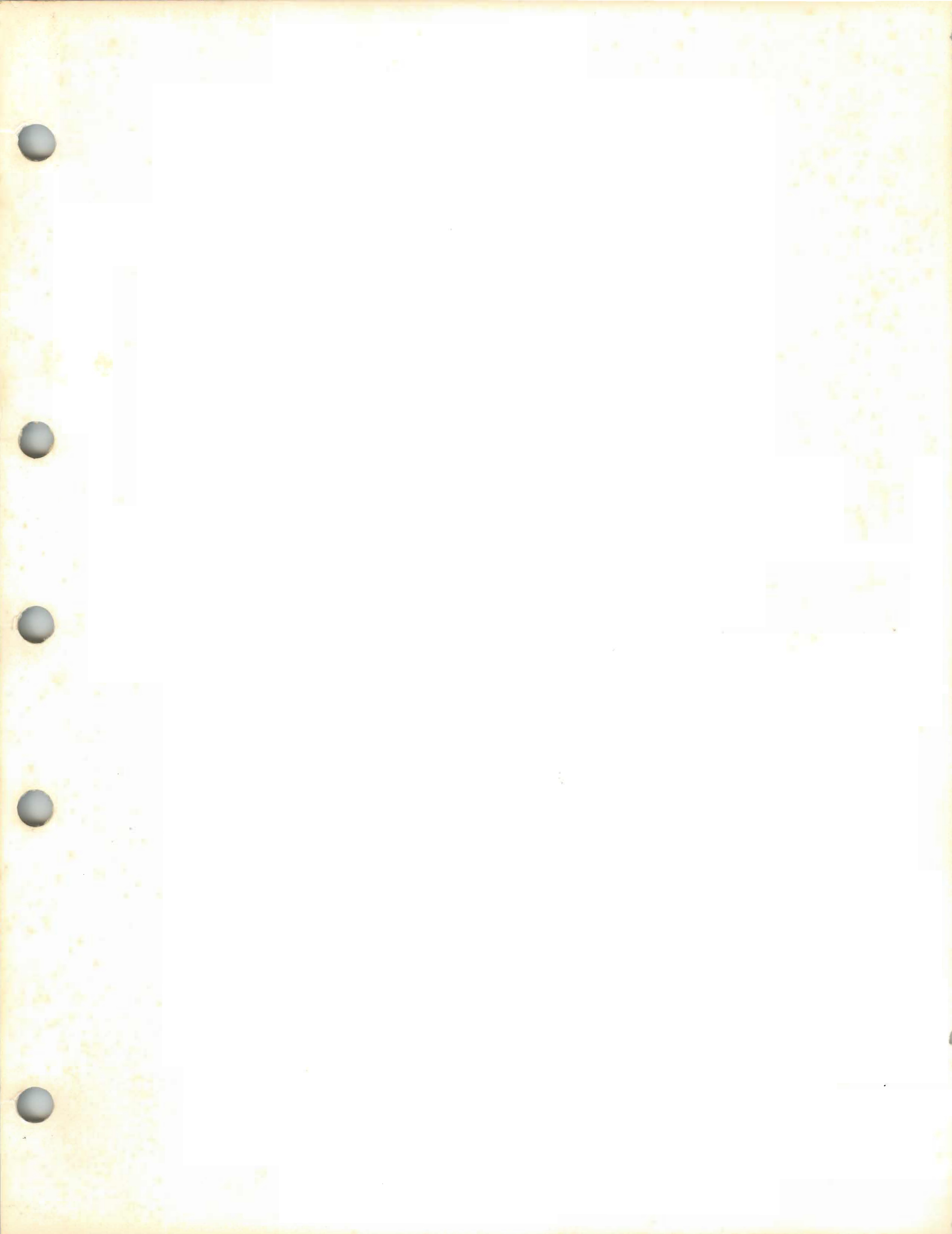
International Business Machines Corporation
Information Development
Department 245
3605 North Hwy 52
ROCHESTER MN 55901-9986

Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold
Along Line



IBM ®

Program Number
5728-ST1

21F2758

SC21-9609-1

Printed in U.S.A.